·开发宝典丛书·

一本百科全书式的Visual C++编程宝典，全面、新颖、详细、深入和实用
资深程序员15年开发经验的总结，完美展现Visual C++的五大应用领域

Visual C++ 编程实战宝典

（33.6小时配套教学视频 + 3小时进阶教学视频）

李琳娜　等编著

- **全面**：全面涵盖Visual C++开发基础知识、界面开发、数据库开发、网络开发、系统开发及多媒体开发等内容
- **新颖**：以当前流行的Visual Studio 2010作为开发环境介绍Visual C++的各项技术
- **详细**：结合图示，从概念、语法、示例、技巧和应用等多角度分析每个知识点
- **实用**：提供了100个综合实例、2个大型项目开发案例、57个实践练习题
- **深入**：剖析了硬件设备控制、系统配置、DDL、多线程等其他图书很少涉及的内容
- **高效**：提供了33.6小时配套教学视频，赠送3小时进阶视频，高效而直观

超值、大容量DVD光盘

- 本书各章涉及的实例源文件
- 33.6小时本书配套教学视频
- 3个项目案例源程序及3小时教学视频
- 324页《C/C++程序员面试宝典》电子书



清华大学出版社

开发宝典丛书

Visual C++编程实战宝典

李琳娜　等编著

清华大学出版社
北　京

内 容 简 介

本书以 Visual Studio 2010 作为开发环境，由浅入深，全面、系统地介绍了 Visual C++开发的各项技术。书中的各个技术点都提供了实例供读者实战演练，各章后还提供了实战练习题帮助读者巩固和提高。**另外，本书配 1 张 DVD 光盘，内容为作者专门为本书录制的 33.6 小时配套教学视频，还收录了本书涉及的所有实例源文件，以帮助读者更加高效、直观地学习本书内容。**

本书共分 7 篇。第 1 篇介绍 Visual Studio 2010 开发环境及搭建、C++基本语法及面向对象思想；第 2 篇介绍 Windows 编程、MFC 基础、菜单、工具栏、状态栏、Windows 标准控件、MFC 类、文档/视图结构、对话框等技术；第 3 篇介绍数据库编程基础及 SQL Server、ADO、ODBC、OLE DB、MySQL 等数据库访问技术；第 4 篇介绍 Windows 套接字编程、邮槽和管道的使用、串行端口编程、Internet 编程等；第 5 篇介绍磁盘操作、系统控制与调用、应用程序的操作、系统工具的操作、桌面的相关操作、系统信息操作、消息的使用、剪贴板的使用、鼠标键盘的操作、操作注册表、读写 INI 文件、读写 XML 文件、动态链接库编程、多线程编程等；第 6 篇介绍文本字体、图形与图像编程、声音与动画编程、DirectX 图形开发等；第 7 篇详细介绍网络音频播放系统、GPS 定位系统项目案例的开发，以提高读者的实战水平。

本书适合所有想全面学习 Visual C++开发技术的人员阅读，也适合用 Visual C++进行开发的工程技术人员和科研人员阅读。对于经常使用 Visual C++做开发的人员，本书是一本不可多得的案头必备参考手册。

图书在版编目（CIP）数据

Visual C++编程实战宝典 / 李琳娜等编著. —北京：清华大学出版社，2014
（开发宝典丛书）
ISBN 978-7-302-34793-4

Ⅰ. ①V…　Ⅱ. ①李…　Ⅲ. ①C 语言－程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2013）第 301372 号

责任编辑： 夏兆彦
封面设计： 欧振旭
责任校对： 徐俊伟
责任印制：

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn, http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 52.75　　**字　　数：** 1317 千字
（附光盘 1 张）
版　　次： 2014 年 8 月第 1 版　　**印　　次：** 2014 年 8 月第 1 次印刷
印　　数：
定　　价： 元

产品编号：056430-01

前　言

Visual C++从字面上理解的意思为可视化 C++编程。它将 C++、Windows API 和 MFC 强强组合。同时，Visual C++也是一种集成开发环境（IDE）。其经典版本为 Visual C++ 6.0。在该 IDE 中，提供了各种高效开发工具和向导，可以极大地提高开发效率。因此它一直都是最为流行的 Windows 开发技术之一，广泛应用于界面开发、数据库开发、网络开发、系统开发和多媒体开发等绝大多数领域。作为 Visual C++开发所用到的核心开发语言 C++，它功能强大，兼容面向过程和面向对象两种编程模式，也是当前最流行的开发语言之一。Windows API 是微软提供的应用程序接口，可以实现开发人员的各种需求。MFC 是为了简化 Windows API 编程而提出的开发框架，可以更高效地开发各类应用程序。所有这些，都构成了 Visual C++开发所必须掌握的几大技术，需要开发人员很好地掌握。

随着各种开发技术的发展和程序复杂度的提高，Visual C++ 6.0 这个经典版本的各种弊端也逐步暴露了出来，严重地影响了程序员的开发工作。例如，它对 C++语言的支持只有 80%左右，它不支持多屏幕开发……。为此，微软提供了更新的版本。

本书便是以微软最新推出的 Visual Studio 2010 为开发环境来介绍 Visual C++的各项开发技术。笔者结合自己多年的 Visual C++开发经验和心得体会，花费了一年多的时间写作本书。希望各位读者能在本书的引领下跨入 Visual C++开发大门，并成为一名开发高手。本书结合大量多媒体教学视频，全面、系统、深入地介绍了 Visual C++开发技术，并以大量实例贯穿于全书的讲解之中，最后还详细介绍了网络音频播放系统和 GPS 定位系统两个项目案例的开发。学习完本书后，读者应该可以具备独立进行项目开发的能力。

本书特色

1．配大量多媒体语音教学视频，学习效果好

作者专门为本书录制了大量的同步配套教学视频辅助学习，以便读者更加轻松、高效地学习。这些视频与本书实例源文件一起收录于本书配套 DVD 光盘中。

2．内容全面、系统、深入

本书介绍了 Visual C++开发的基础知识、界面开发、数据库开发、网络编程、系统功能编程和多媒体开发等内容，最后还详细介绍了两个项目案例的开发。

3．讲解由浅入深、循序渐进，适合各个层次的读者阅读

本书从 Visual C++的基础开始讲解，逐步深入到 Visual C++的高级开发技术及应用。书中内容梯度从易到难，讲解由浅入深、循序渐进，适合各个层次的读者阅读，相信读者

均有所获。

4. 贯穿大量的开发实例和技巧，迅速提升开发水平

本书在讲解知识点时贯穿了大量短小精悍的典型实例，并给出了大量的开发技巧，以便让读者更好地理解各个概念和开发技术，体验实际编程，迅速提高开发水平。

5. 详解典型项目案例开发，提高实战水平

本书详细介绍了网络音频播放系统和GPS定位系统项目案例的开发。通过这两个项目案例的讲解，可以提高读者的软件项目开发水平，从而具备独立进行项目开发的能力。

6. 提供技术支持，答疑解惑

读者在阅读本书时有任何疑问都可以发电子邮件到 book@wanjuanchina.net 或者 bookservice2008@163.com 以获得帮助。读者也可以在本书的技术论坛上留言，会有专人负责答疑。论坛网址 http://www.wanjuanchina.net。

本书内容及体系结构

第1篇　Visual C++开发基础（第1～4章）

本篇主要内容包括 Visual Studio 2010 集成开发环境的搭建、Visual Studio 2010 基本应用程序的创建、C++语言基础、C++面向对象程序设计等。通过本篇的学习，读者可以掌握 Visual Studio 2010 开发环境和 C++编程的语法及核心思想。

第2篇　界面开发（第5～10章）

本篇主要内容包括 Windows 编程、MFC 基础、菜单、工具栏、状态栏、Windows 标准控件、MFC 常用类、文档/视图结构、对话框等内容。通过本篇的学习，读者可以掌握 Visual C++界面编程的核心技术与应用。

第3篇　数据库开发（第11～15章）

本篇主要内容包括数据库编程基础、SQL Server 数据库基础、ADO 数据库访问技术、ODBC 数据库访问技术、OLE DB 数据库访问技术、MySQL 数据库访问技术等。通过本篇的学习，读者可以掌握 Visual C++中各种常见的数据库访问技术。

第4篇　网络编程（第16～19章）

本篇主要内容包括 Windows 套接字编程、邮槽和管道的使用、串行端口通信编程、Internet 编程等。通过本篇的学习，读者可以掌握 Visual C++中有关网络通信编程的核心技术及应用。

第5篇　系统编程（第20～23章）

本篇主要内容包括磁盘操作、系统控制与调用、应用程序的操作、系统工具的操作、

桌面的相关操作、系统信息操作、消息的使用、剪贴板的使用、鼠标键盘的操作、操作注册表、读写 INI 文件、读写 XML 文件、动态链接库编程、多线程编程等。通过本篇的学习，读者可以掌握 Visual C++中有关系统功能编程的核心技术及应用。

第6篇　多媒体开发（第24～27章）

本篇主要内容包括文本字体、图形与图像编程、声音与动画编程、DirectX 图形开发等。通过本篇的学习，读者可以掌握 Visual C++中有关多媒体开发的核心技术及应用。

第7篇　项目开发实战（第28、29章）

本篇主要内容包括网络音频播放系统项目案例开发和 GPS 定位系统项目案例开发。通过本篇的学习，读者可以全面应用前面章节所学的开发技术进行软件项目开发，达到可以独立开发项目的水平。

本书超值 DVD 光盘内容

- 本书各章涉及的实例源文件；
- 33.6 小时本书配套教学视频；
- 3 个 Visual C++项目案例源程序及 3 小时教学视频；
- 324 页《C/C++程序员面试宝典》电子书。

本书读者对象

- Visual C++初学者；
- 想全面学习 Visual C++开发技术的人员；
- Visual C++专业开发人员；
- 利用 Visual C++进行开发的工程技术人员；
- Visual C++开发爱好者；
- 大中专院校的学生；
- 社会培训班学员；
- 需要一本案头必备手册的程序员。

本书阅读建议

- 建议没有基础的读者，从前往后阅读，尽量不要跳跃。
- 书中的实例和示例建议读者都要亲自上机动手实践，学习效果会更好。
- 学习每章内容时，建议读者先仔细阅读书中的讲解，然后再结合本章教学视频，学习效果会更佳。

本书作者及编委会成员

本书由李琳娜主笔编写。其他参与编写的人员有陈虹翔、陈慧、陈金枝、陈勤、季永

辉、雷双社、李加爱、李兴南、林天云、刘升华、柳刚、罗永峰、吕琨、马娟娟、潘玉亮、齐凤莲、秦光、秦广军、邵国红、宋敬彬、孙海滨、索依娜、王敏、王欣惠、王秀明、王秀萍、魏星、吴宝生、伍远明、谢平。

本书的编写对笔者而言是一个“浩大的工程”。虽然笔者投入了大量的精力和时间，但只怕百密难免一疏。若读者在阅读本书时发现任何疏漏，希望能及时反馈给我们，以便及时更正。

最后祝各位读者读书快乐，学习进步！

编著者

目　　录

第 1 篇　Visual C++开发基础

第2篇 界面开发

第 3 篇 数据库开发

第 4 篇 网 络 编 程

第 5 篇 系 统 编 程

第 6 篇 多媒体开发

第 7 篇 项目开发实战

第 1 篇　Visual C++开发基础

第 1 章　Visual Studio 2010 集成开发环境

Visual Studio 是微软公司推出的集成开发环境。它是目前最流行的 Windows 平台应用程序开发环境。而 Visual Studio 2010 的开发环境界面相较 Visual Studio 2008 被重新设计和组织，所以也变得更加简单明了。工欲善其事，必先利其器。本章就详细讲述 Visual Studio 2010 的集成开发环境。

1.1　Visual Studio 2010 及其开发环境

Visual Studio 2010 包含了 Visual C++的集成开发环境，它将 C++程序的编辑、编译和调试等功能集成在一起，同时提供了 MFC、ATL 等框架，读者使用此开发工具可以有效地提高开发效率。本书以 Visual Studio 2010 旗舰版为例介绍其开发环境。

1.1.1　Visual Studio 2010 的安装

要在 Visual Studio 2010 环境下进行开发，首先需要在 Windows 系统下安装 Visual Studio 2010。安装 Visual Studio 2010 的过程与安装其他常用工具软件的过程是非常相似的，都以向导的形式指导用户安装。其过程如下。

（1）双击安装程序 Setup.exe，弹出欢迎对话框，单击“下一步”按钮，弹出许可确认对话框，如图 1-1 所示。

（2）选择“我已阅读并接受许可条款”单选按钮，单击“下一步”按钮，选择要安装的功能，如图 1-2 所示。

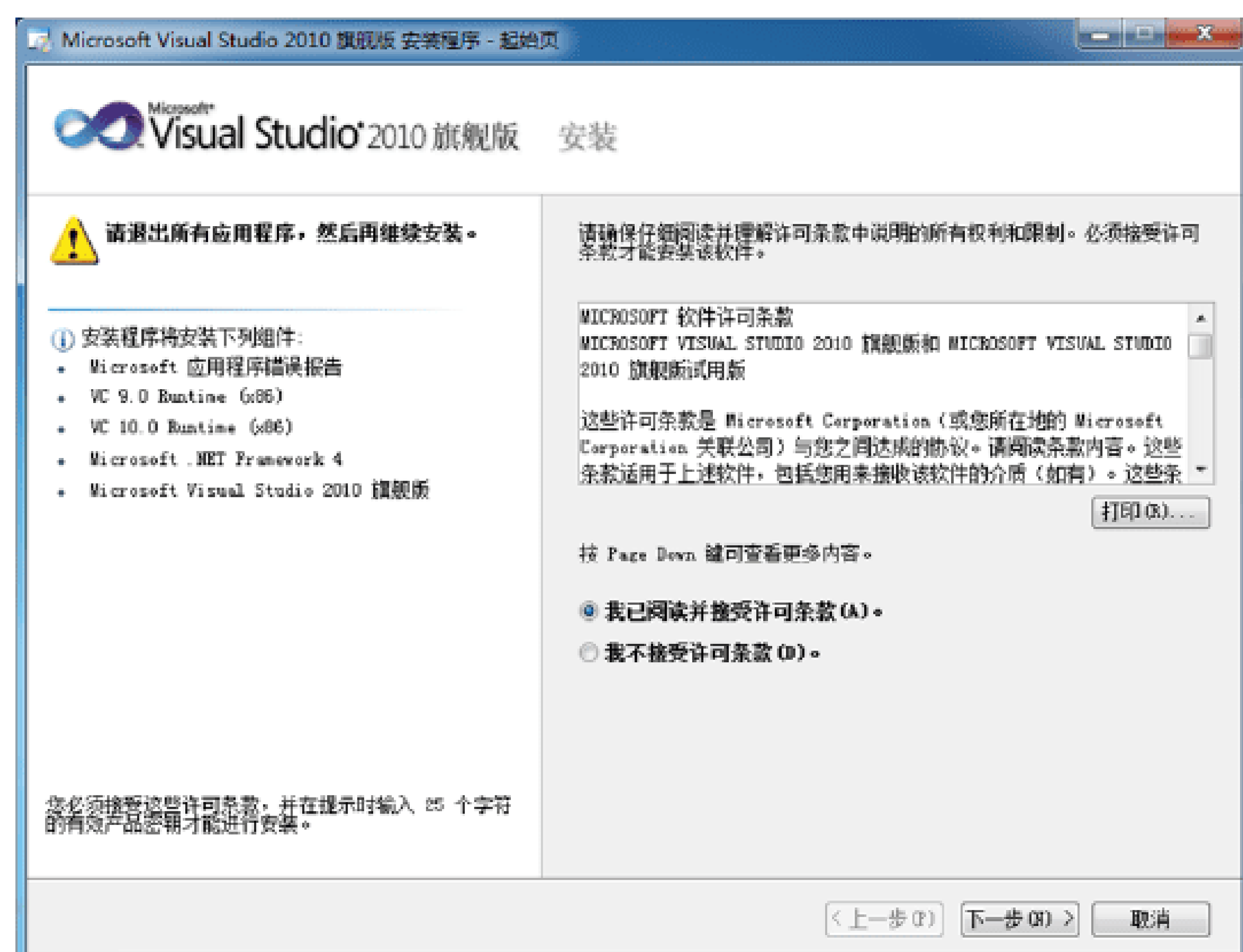


图 1-1　Visual Studio 2010 安装——许可确认

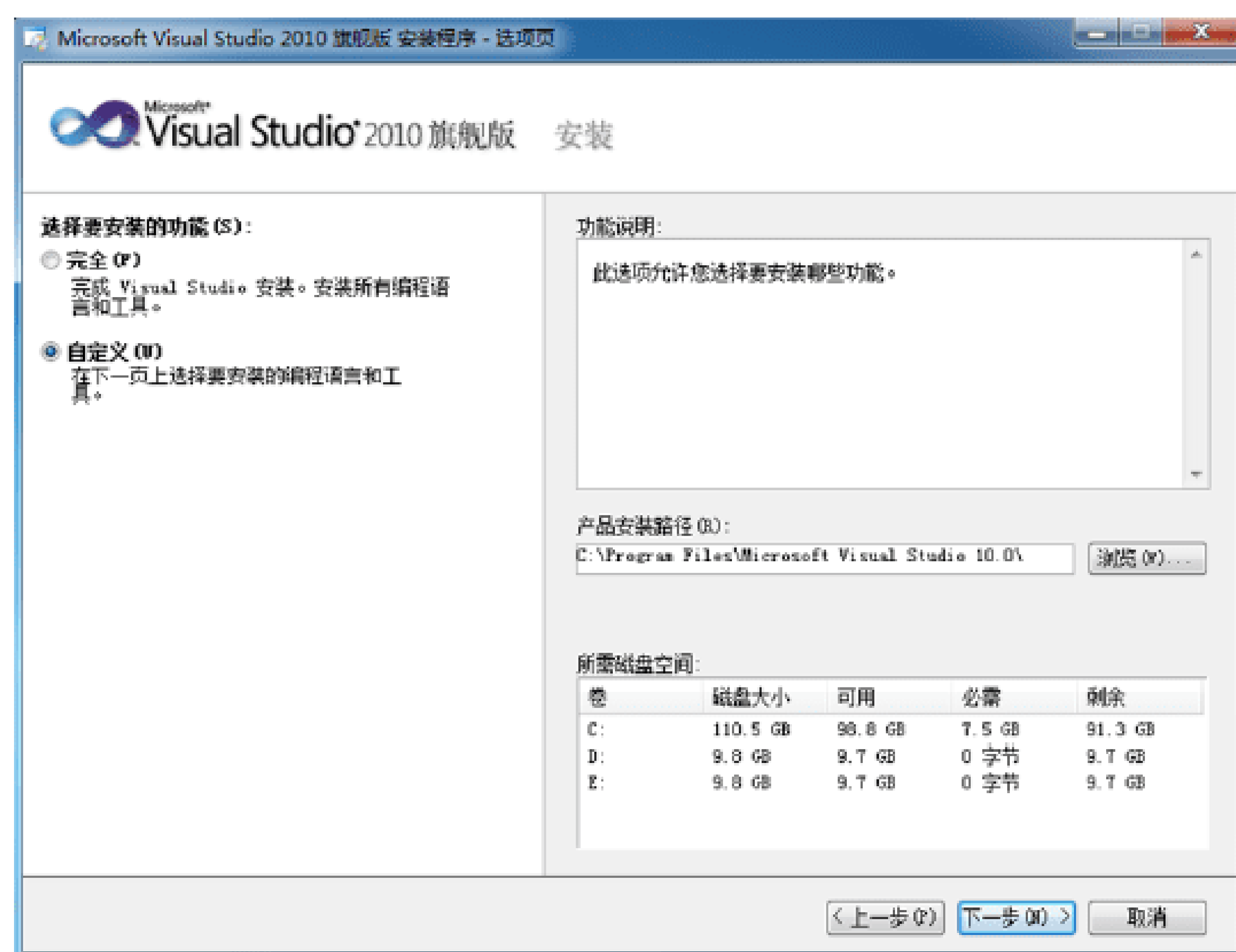


图 1-2　Visual Studio 2010 安装——选择要安装的功能

（3）选择“自定义”单选按钮，单击 “下一步”按钮，然后在对话框中选择要安装的功能，如图 1-3 所示。在选择了需要的功能后再单击“安装”按钮，开始安装，如图 1-4 所示。

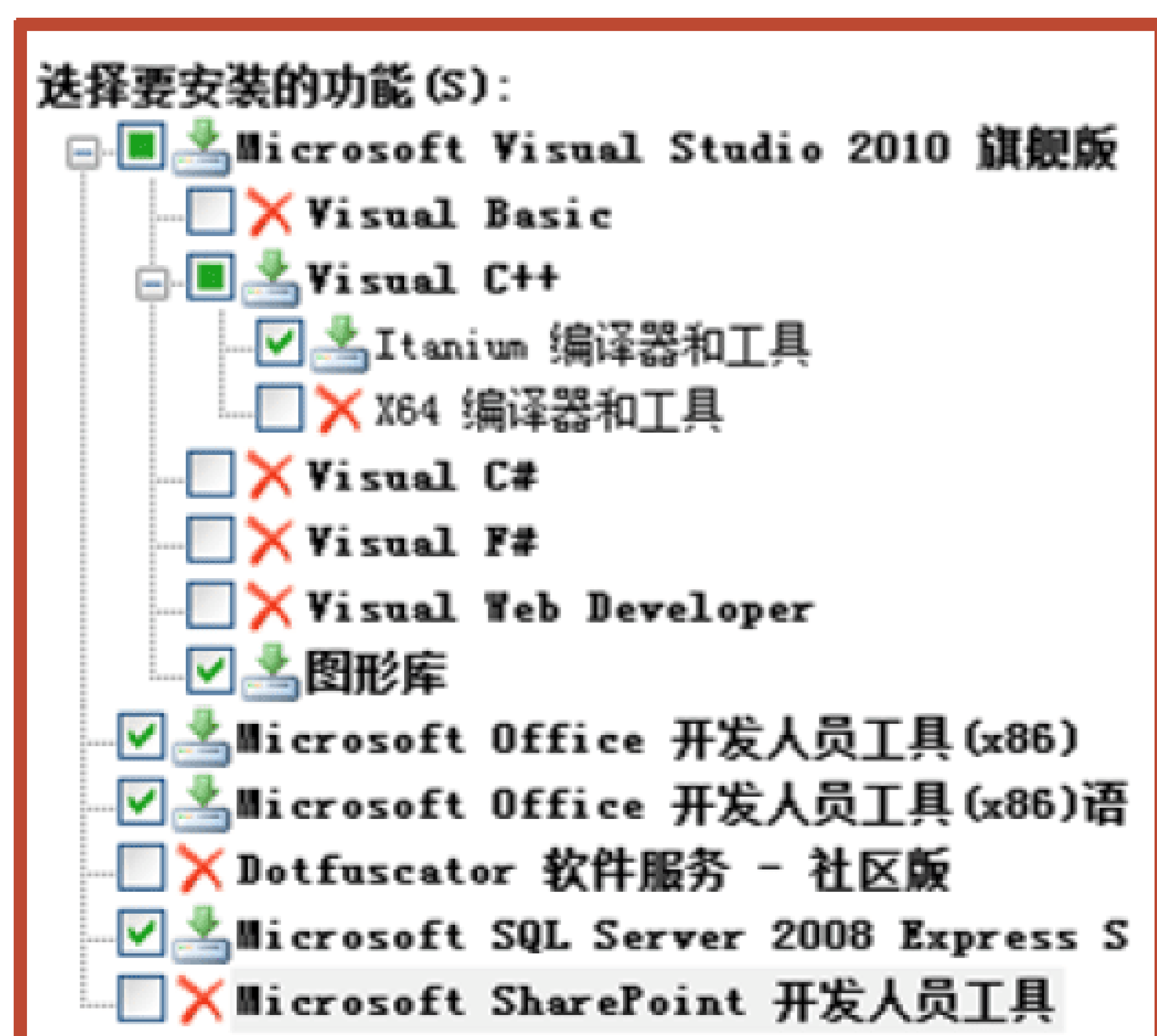


图 1-3　Visual Studio 2010 安装——安装功能选择

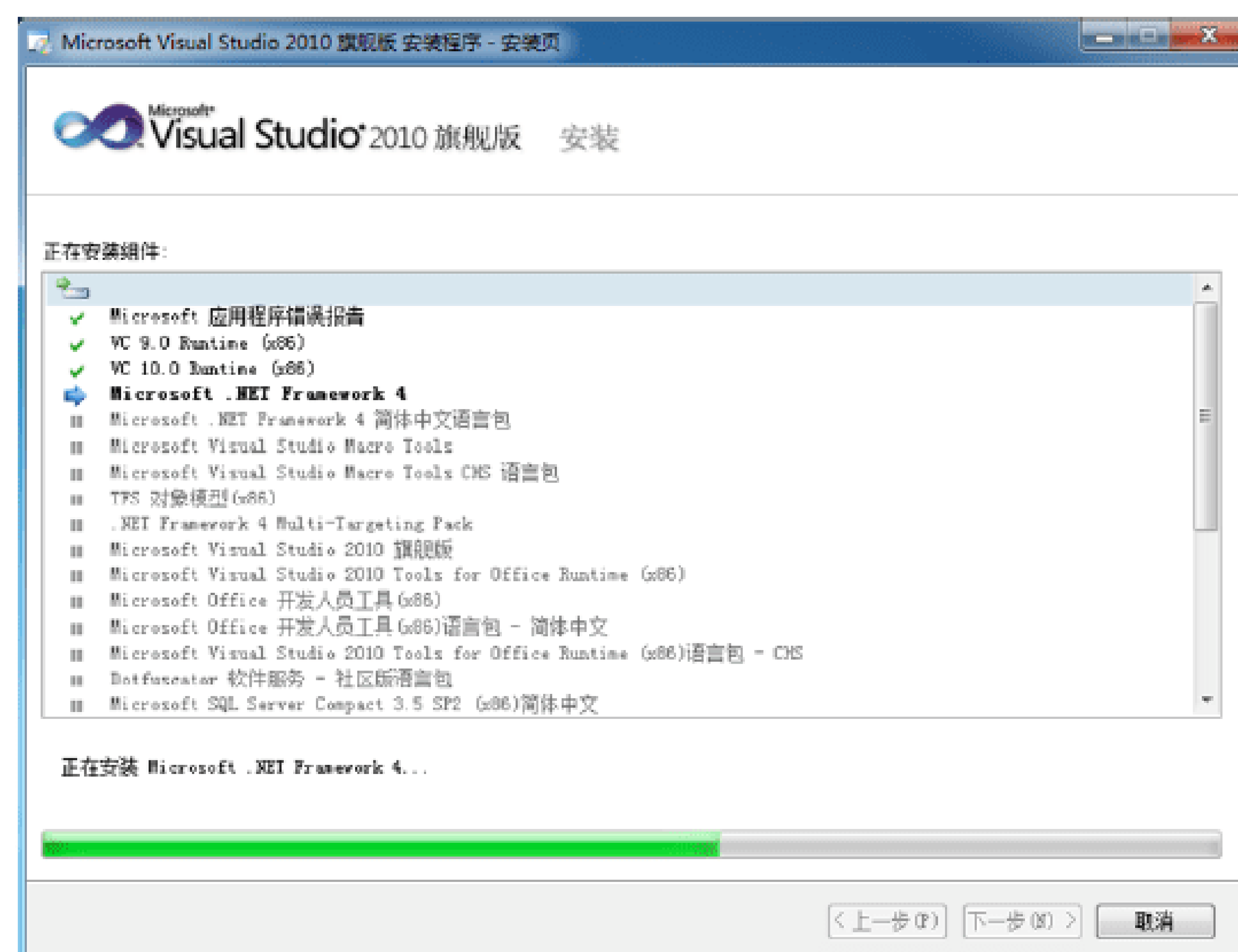


图 1-4　Visual Studio 2010 安装——开始安装

（4）安装的过程比较耗时，一段时间后，对话框会显示安装成功，如图 1-5 所示。读者可以单击窗体上的“安装文档”按钮来安装帮助文档，也可以单击“完成”按钮，结束安装过程。

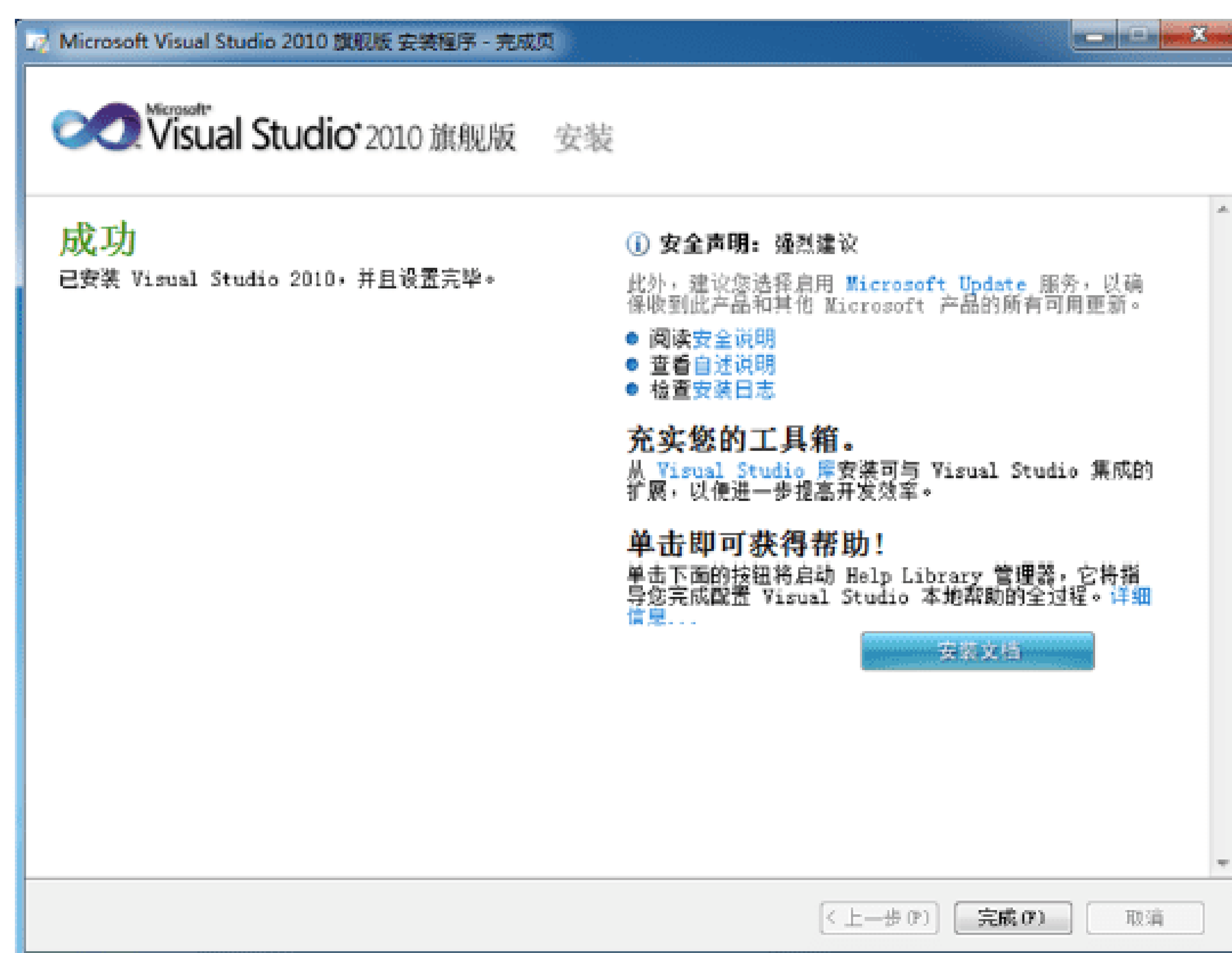


图 1-5　Visual Studio 2010 安装——安装成功

1.1.2　Visual Studio 2010 开发环境

Visual Studio 2010 的开发环境包括标题栏、工具栏、菜单栏、视图区、状态栏、输出窗口和编辑区，如图 1-6 所示。

标题栏是 Visual Studio 2010 的标题显示区，包括 Visual Studio 2010 的 Logo、当前操作的解决方案或项目的名称，以及最小化 Visual Studio 2010 按钮、还原/最大化 Visual Studio 2010 按钮和关闭按钮。

菜单栏由多个菜单组成，每个菜单又包含子菜单和多个菜单项。Visual Studio 2010 就

是通过开发人员使用这些菜单项，执行功能来实现可视化程序开发。



图 1-6　Visual Studio 2010 的开发环境

工具栏是具有相同功能的多个菜单项组成的命令栏，其中的工具按钮与菜单栏中的菜单项的功能是相同的，Visual Studio 2010 中包含多个工具栏，如编辑工具栏、SQL 工具栏和文件工具栏等。在本质上，菜单栏的菜单项和工具栏的工具按钮的核心是相同的，只是表现形式不同。

状态栏是 Visual Studio 2010 工作状态的显示区，其主要显示消息和一些有用的信息，如当前正在操作的内容所在的位置、系统当前时间等。

工作区是用户使用 Visual Studio 2010 的主要工作区域，其主要包括视图区、输出窗口和编辑区等。视图区用于显示类的信息、文件信息和资源信息。输出区则显示程序编译、链接和生成信息以及查找结果和 SQL 执行结果等操作输出信息。编辑区用于存放编辑器，编辑器用于进行源代码、资源等的编辑。

1.1.3　Visual Studio 2010 向导

Visual Studio 2010 除了提供了可视化的开发环境外，还为用户提供了各种向导，大大加速了应用程序的开发过程。依次选择“文件”|“新建”|“项目”菜单项，可以打开“新建项目”对话框，如图 1-7 所示。此向导可以依据用户对模板的不同选择，创建出不同的模板程序。

类向导，是针对 MFC 和 ATL 库的。使用类向导可以更简单快捷地编写类，如创建新类、定义消息句柄及重载虚函数，并能从对话框、窗体视图和记录视图的控件中搜集数据、为对象增加属性、方法和事件等，这些功能使用类向导来实现可以大大减少代码的输入量。在 Visual Studio 2010 中，选择“项目”|“类向导”命令或者使用 Ctrl+Shift+X 快捷键即可调用类向导，如图 1-8 所示。

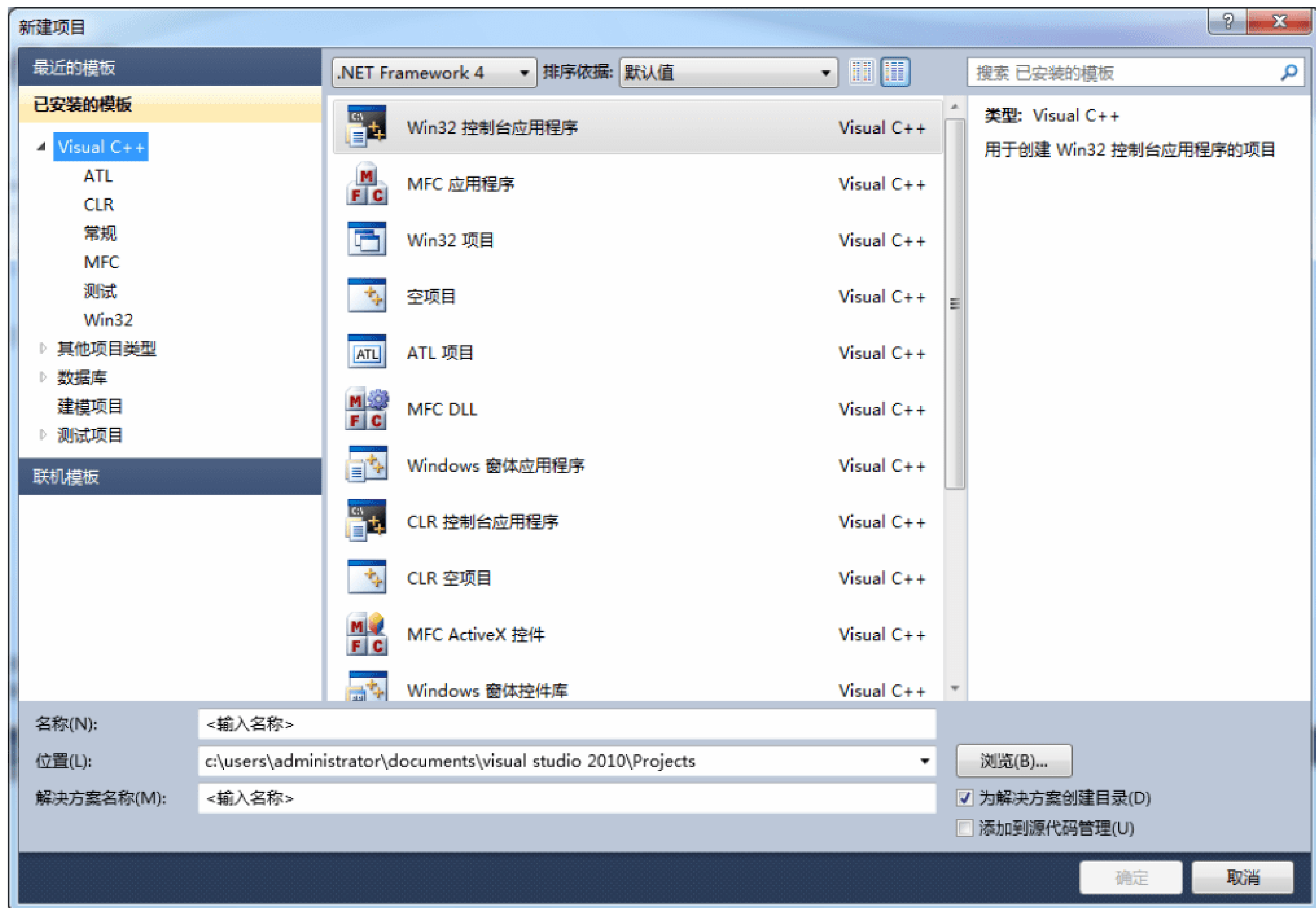


图 1-7 “新建项目”对话框

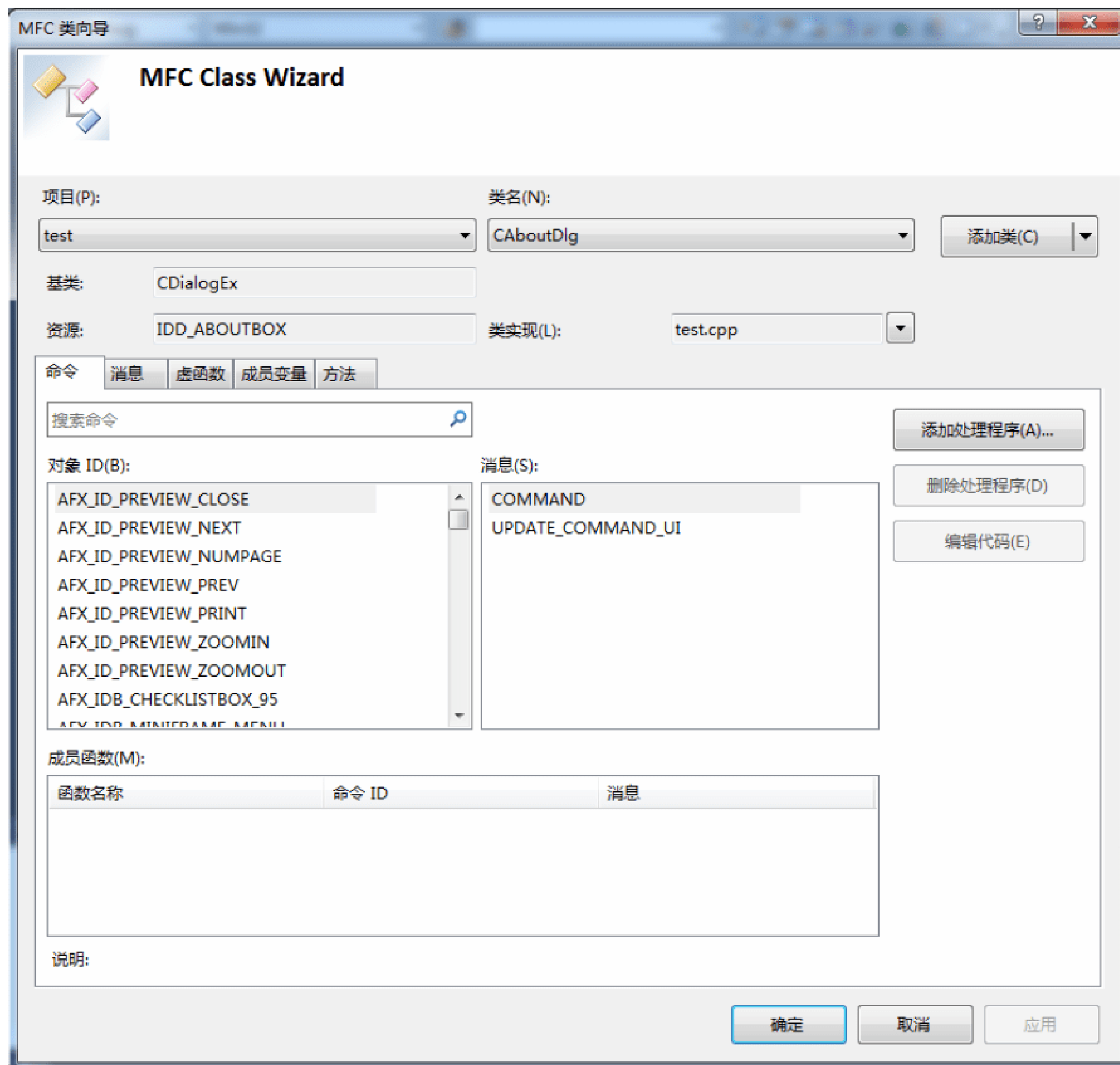


图 1-8　Visual Studio 2010 类向导

1.2　工作区视图

Visual Studio 2010 中提供了多种视图，从各种不同的角度展示了项目的结构，从而帮助开发人员从各种角度深刻理解项目。下面介绍有关工作区视图的内容。

1.2.1　解决方案视图

Visual Studio 2010 中的解决方案视图，用于显示当前项目或工作区中包含的文件。单击选择指定元素，则可以快速地定位到选择的文件中，如图 1-9 所示。

1.2.2　类视图

Visual Studio 2010 中的类视图，用于显示当前项目或工作区中包含的类定义。选择其中的元素，则可以快速地定位到定义上，如图 1-10 所示。

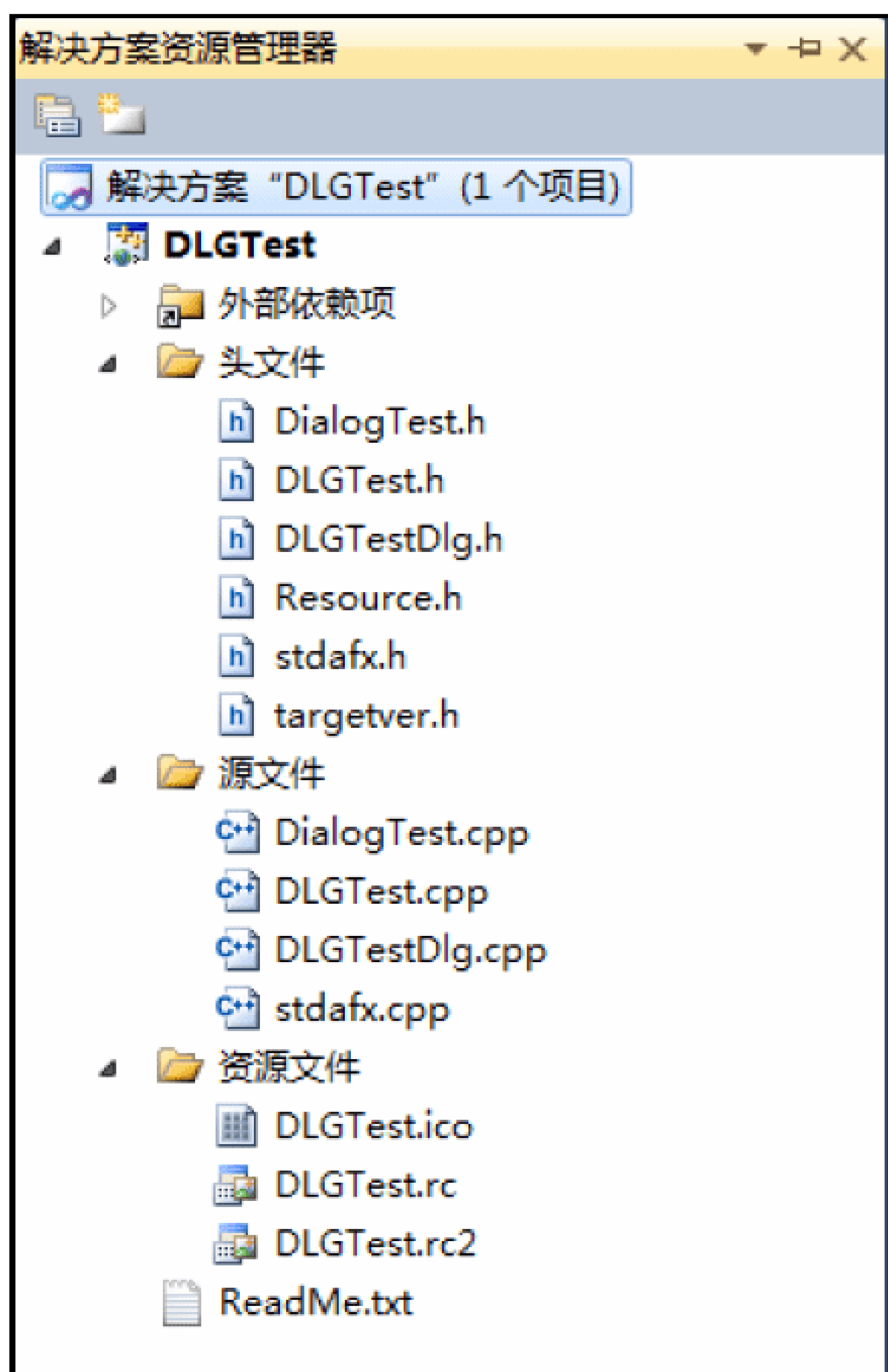


图 1-9　解决方案视图

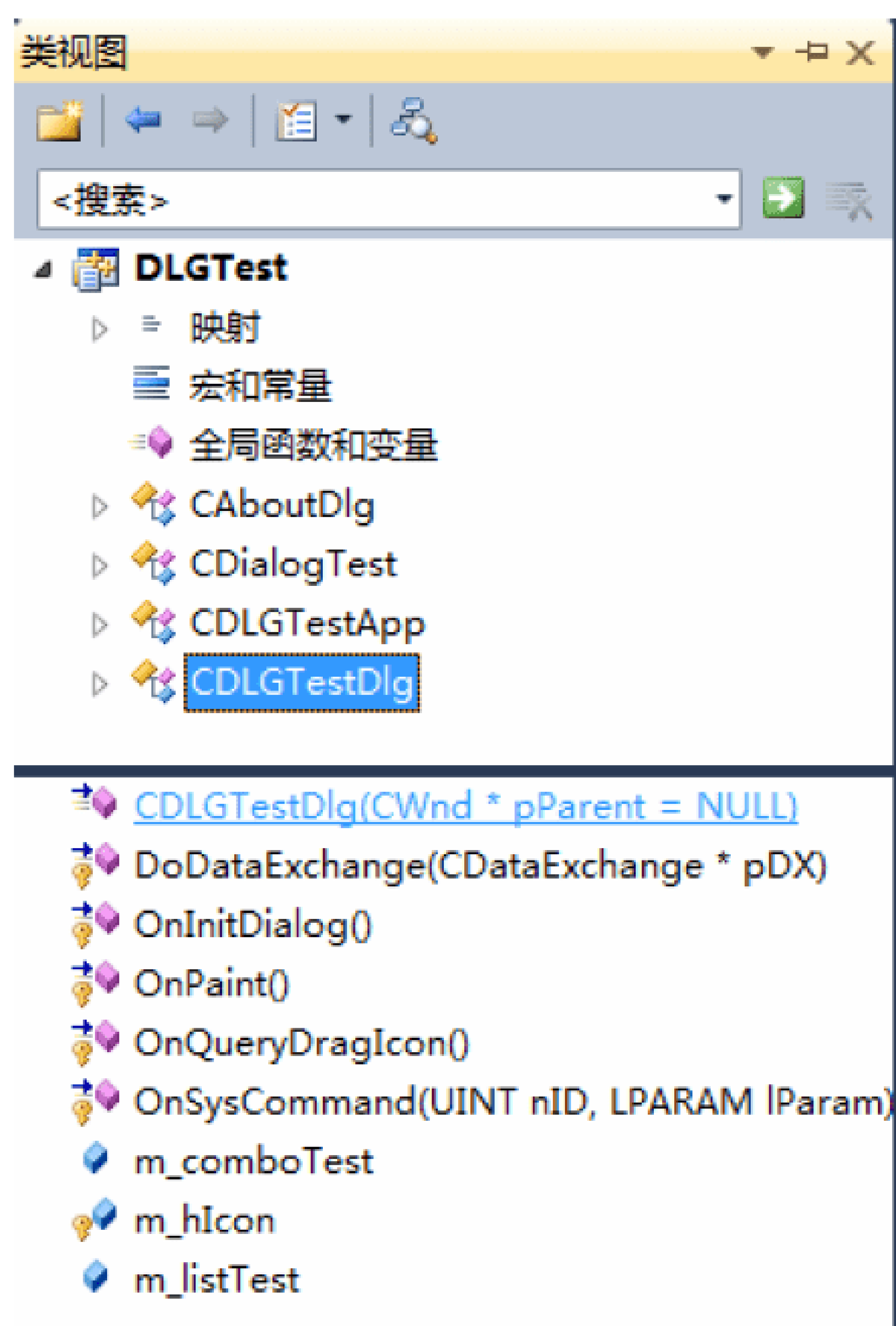


图 1-10　类视图

1.2.3　资源视图

Visual Studio 2010 中的资源视图，用于显示当前项目或工作区中包含的资源。选择其中的元素，则可以快速地定位到资源上，如图 1-11 所示。

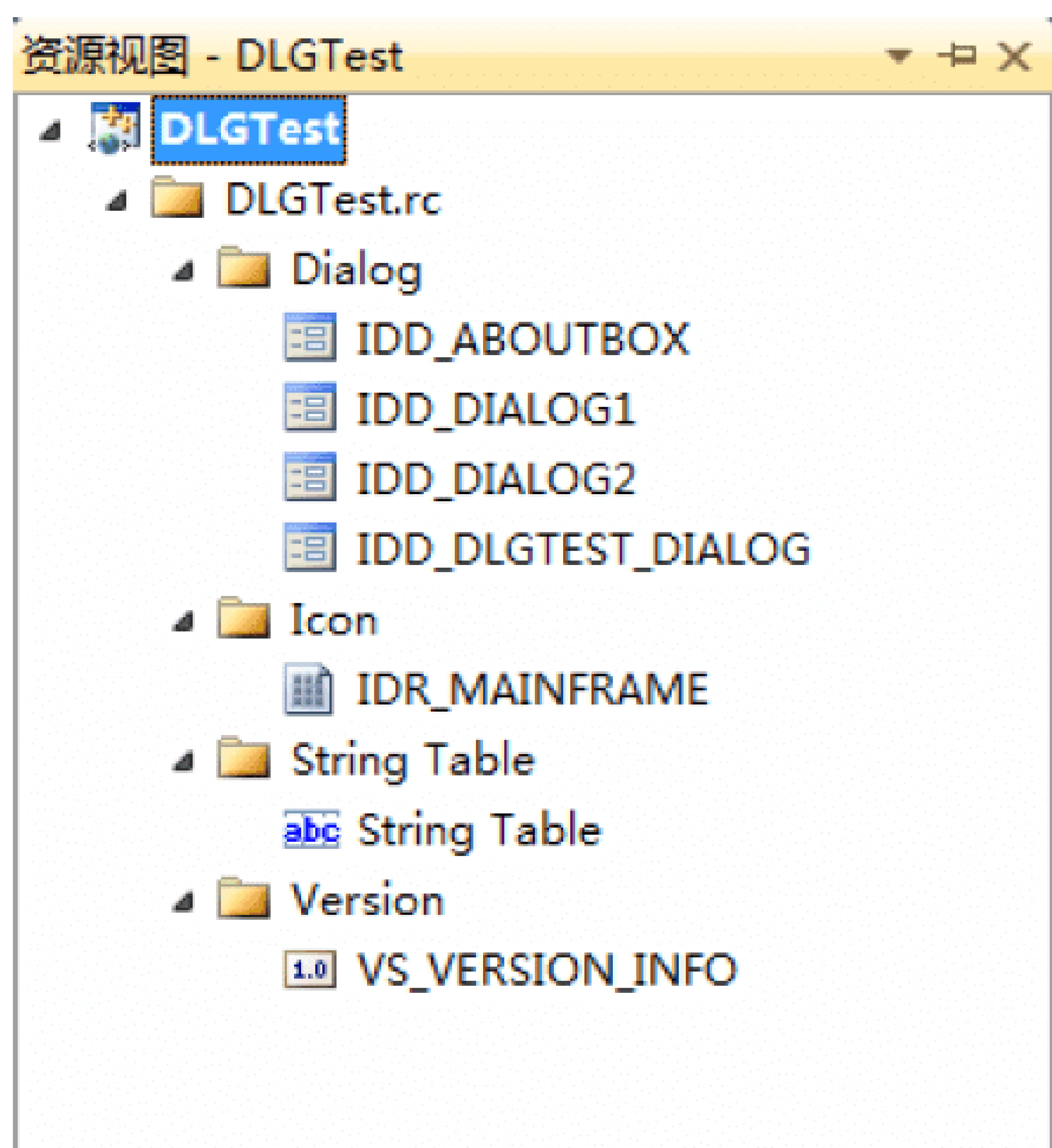


图 1-11　资源视图

1.3　资源与资源编辑器

资源编辑器可以快速方便地创建和编辑应用程序资源。资源编辑器也可以用来创建新资源、修改现有的资源、复制现有的资源和删除旧资源。虽然 Visual Studio 2010 包含多种资源编辑器，但是它们的使用方式都是一致的。本节就分别介绍 Visual Studio 2010 中支持的资源及其相应的资源编辑器的使用。

1.3.1　资源的类型

在 Visual Studio 2010 中，包含多种项目资源，为项目提供各种附加功能，具体如下所述。

- 加速键资源：表示工程中的加速键。
- 对话框资源：表示工程中的对话框，是工程中比较重要的一种资源，大部分程序可以以对话框的形式提供用户界面。
- 图形资源：表示工程中的图形，包括程序图标等与图形有关的资源。
- HTML 资源：表示工程中使用的 HTML 文件。
- 菜单资源：表示工程中的菜单，可以使用它定制符合程序的菜单。
- 字符串资源：表示工程中用到的字符串，使用它可以将工程中的字符串用标识符代替。当用户需要更改字符串的内容时，只需要修改字符串资源即可，不需要到程序中查找修改。尤其在多语言程序时，字符串资源非常有用。
- 工具栏资源：表示工程中的工具栏，可以使用它定制符合程序的工具栏。
- 版本信息资源：用于表示工程中的版本信息。

双击某种资源，系统即会在相应的资源编辑器中打开资源。如对话框资源会在对话框编辑器中打开，菜单资源会在菜单编辑器中打开。

1.3.2　资源编辑器

Visual Studio 2010 中提供了资源编辑器来进行资源的编辑。在其中可以创建默认资源，或者是使用资源模板创建资源、修改资源和删除资源。创建默认资源只需要单击资源工具栏中相应的工具按钮即可。从模板文件创建新资源的具体步骤如下。

（1）选择“项目”|“添加资源”命令，打开“添加资源”对话框，如图 1-12 所示。

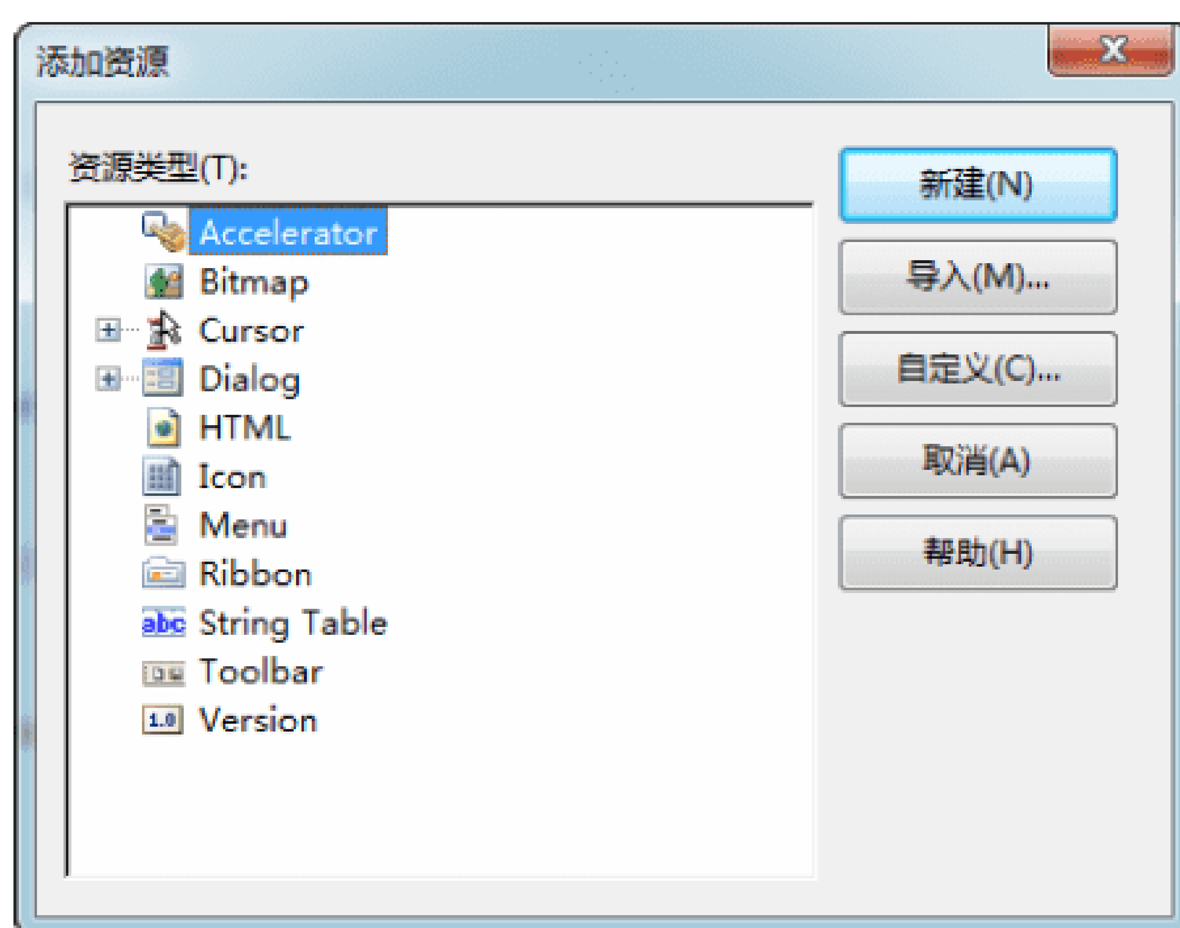


图 1-12　添加资源

（2）从“资源类型”列表框中选择要添加的资源，并单击“新建”按钮。

当创建完资源后，系统会自动为它分配一个唯一的标识符名称和值。如果想要改变标识符值，在资源的属性页面中修改 ID 即可。

1.4　本 章 小 结

本章主要介绍了 Visual Studio 2010 的安装过程、开发环境和向导。通过本章的学习，应该重点掌握 Visual Studio 2010 的安装和开发环境的使用。Visual Studio 2010 向导的使用是本章的难点。第 2 章将以实例为基础介绍如何在 Visual Studio 2010 中创建几种基本的应用程序。

1.5　习　　题

熟悉 Visual Studio 2010 的各个菜单及其菜单项，完成以下操作：

（1）假如解决方案视图、类视图和资源视图都被关闭了，如何通过菜单项来“找回”它们，即再次显示相应视图于窗体之中。

（2）Visual Studio 2010 提供了修改字体类型、字体大小和代码编辑区背景色的功能，将它们分别修改为 Consolas（一种字体）、“12”和“青色”。

（3）默认代码编辑区是不添加行号的，尝试修改设置，为代码编辑区添加行号。

【思路】花时间熟悉一下 Visual Studio 2010 的各个菜单项，了解它们的功能。

第 2 章　Visual Studio 2010 基本应用程序的创建

第 1 章对 Visual Studio 2010 的开发环境做了简要的介绍，从本章开始介绍使用 Visual Studio 2010 进行实际应用程序开发的知识。本章将介绍使用 Visual Studio 2010 的项目向导创建应用程序的基本步骤，并结合示例介绍使用向导如何创建 Win32 控制台应用程序和 MFC 应用程序。

2.1　使用 AppWizard 生成项目

第 1 章介绍过，Visual Studio 2010 为开发人员提供了多种向导，其中应用向导（AppWizard）可以帮助开发人员快速地创建各种不同类型的应用程序，简化相同类型项目的开发工作量。本节将介绍如何使用 AppWizard 生成项目。

2.1.1　解决方案与项目

Visual Studio 2010 使用解决方案和项目组织程序。每个独立的应用程序或模块，都可以看作一个项目，多个相关联或完成同一个目标的项目，可以放在同一个解决方案中。解决方案是项目的容器，可以包含多个项目，这些项目可以是独立的，也可以是相互关联的父子项目或依赖项目。解决方案包含的元素如表 2-1 所示。

表 2-1　解决方案包含的元素

元素名称	扩展名	元 素 内 容
解决方案文件	.sln	记录着解决方案中的项目信息
选项文件	.suo	记录着应用于该解决方案的用户选项
源文件	.cpp	存放程序元素定义的源代码文件
头文件	.h	存放程序元素声明的头文件
资源文件	.rc	存放程序资源的文件

2.1.2　使用 AppWizard 创建项目

为了简化开发，Visual Studio 2010 提供了 AppWizard，即应用程序向导。它为用户预

定义了多种项目模板，使用这些模板，可以快速地创建类似的项目。如使用对话框应用向导，可以创建对话框应用程序所需的基本文件，用户只需要按照步骤进行简单的操作，即可创建基于对话框的应用程序的雏形。使用 AppWizard 向导创建项目的步骤如下。

（1）选择“文件”|“新建”|“项目”命令，打开“新建项目”对话框。

（2）选择 Visual C++，如图 2-1 所示。

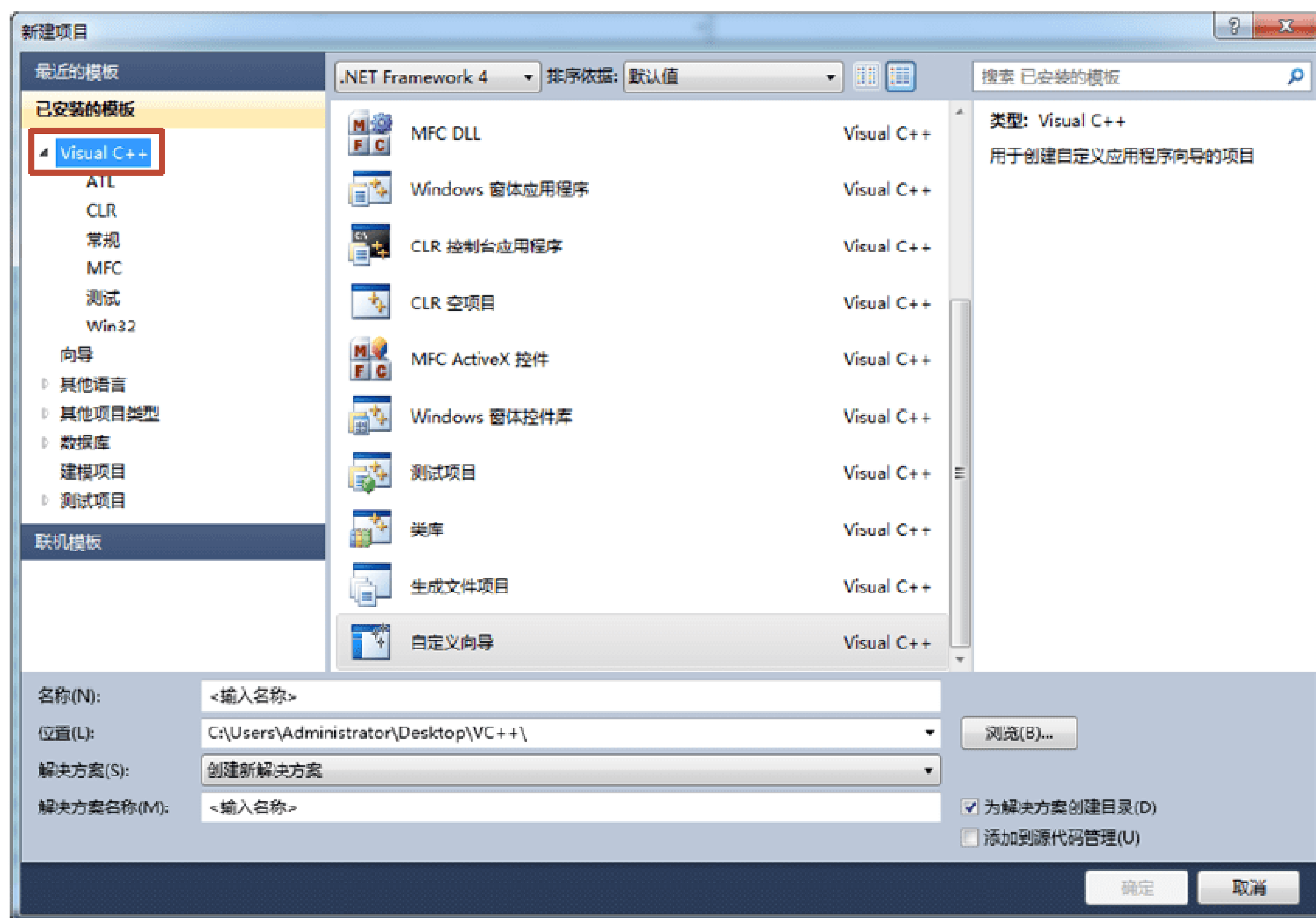


图 2-1　“新建项目”对话框

（3）在中间的项目模板中选择要创建的项目类型。

（4）在下面的“名称”文本框中输入项目名称。

（5）在“位置”文本框中输入项目存放的路径。单击右边的“浏览”按钮，打开如图 2-2 所示的“项目位置”对话框，选择项目存放的路径，并单击“选择文件夹”按钮。

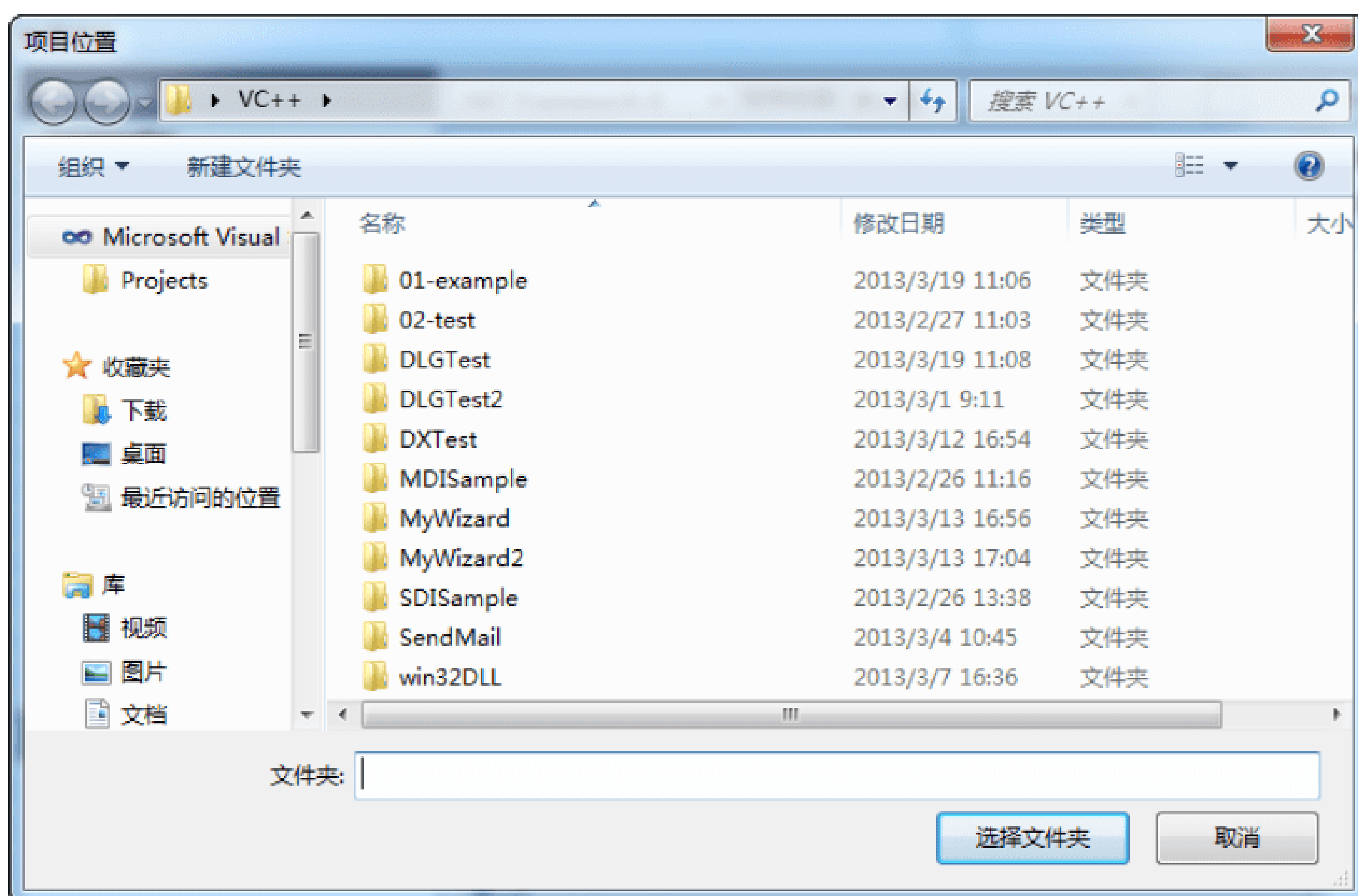


图 2-2　“项目位置”对话框

（6）单击“新建项目”对话框上的“确定”按钮，向导会根据用户选择的项目类型，使用不同的向导引导用户创建新项目。

Visual Studio 2010 的应用向导提供了多种项目类型的向导，创建的步骤都是类似的，只是在第（3）步的选择项目类型时，需要根据情况选择合适的项目类型，向导会根据选择的项目类型使用不同的向导。具体的向导步骤内容会在后面陆续涉及到。

2.2　Win32 控制台应用程序

在 VC 中，一种典型的应用程序是控制台应用程序。它使用控制台 API 函数和标准的 I/O 函数，在控制台对话框中提供对字符的支持，实现与用户间的信息交互。本节将介绍有关 Win32 控制台应用程序的开发步骤。

2.2.1　使用向导生成 Win32 控制台项目

控制台应用程序是使用控制台 API 函数的程序，它提供对控制台对话框中的字符模式的支持。VC 在运行时库也提供对标准 I/O 函数，如 printf()函数和 scanf()函数的支持，实现从控制台输出和输入信息。在 Visual Studio 2010 中，可以使用 AppWizard 生成 Win32 控制台项目。具体步骤如下。

使用 AppWizard 创建项目，在项目列表中选择“Win32 控制台应用程序”项目类型，单击“确定”按钮。打开如图 2-3 所示的 Win32 应用程序向导。正文显示的是目前项目的设置信息，若需要改动可以单击向导左侧的“应用程序设置”，进入应用程序设置页面，如图 2-4 所示。

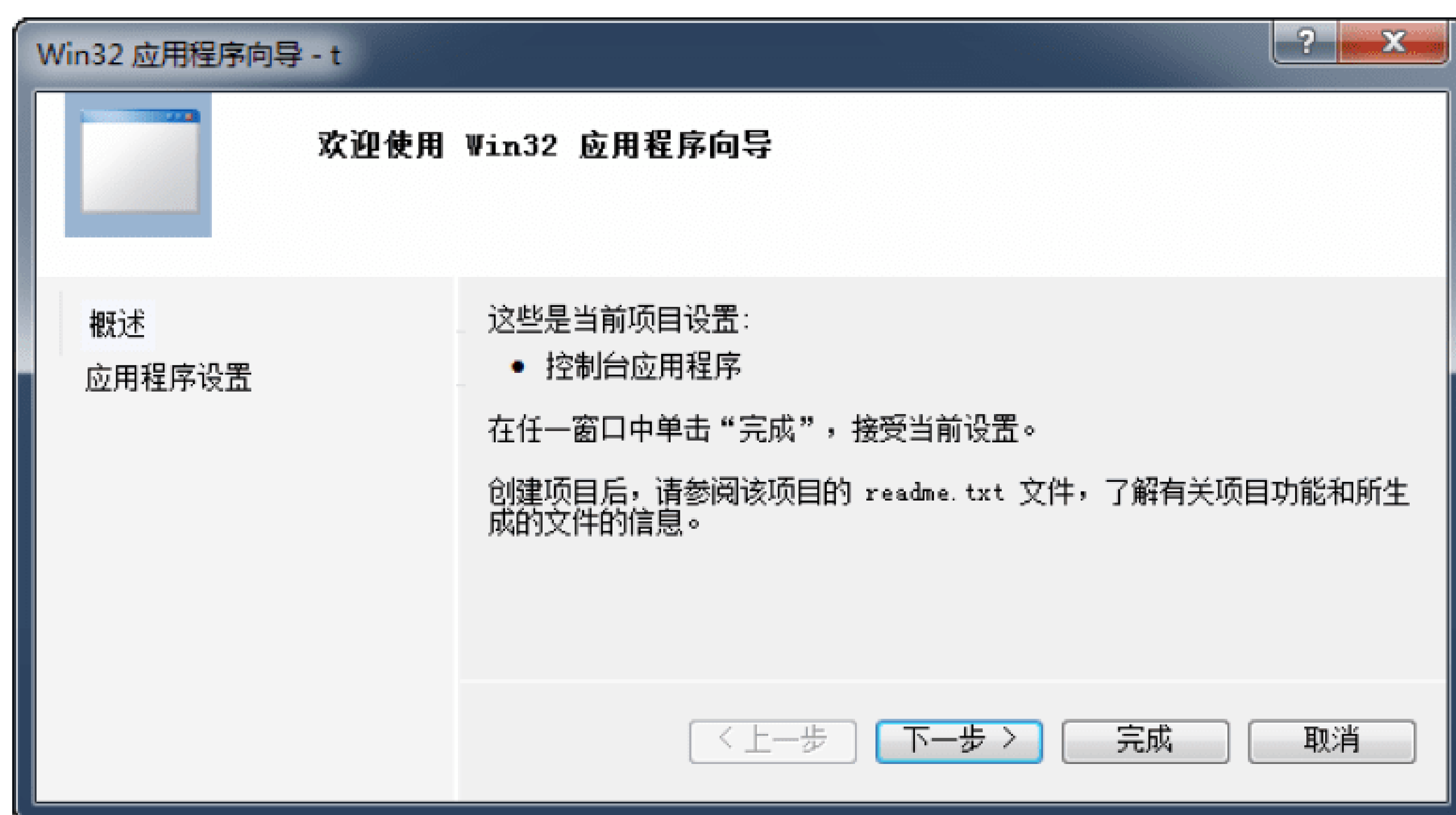


图 2-3　Win32 应用程序向导

2.2.2　添加源文件

创建好项目后，就可以增加实现功能的源文件，步骤如下。

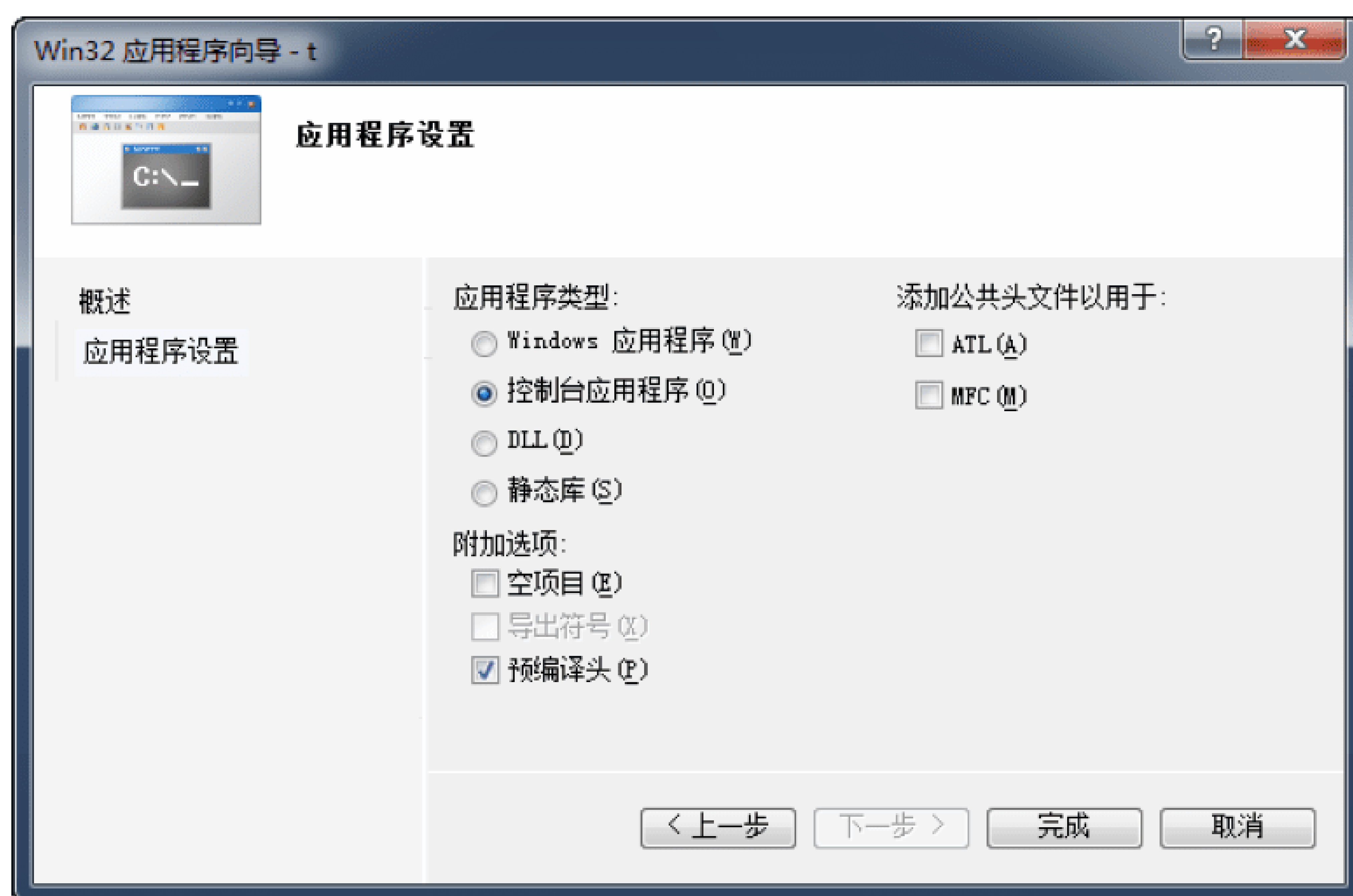


图 2-4　应用程序设置

（1）选择“项目”|“添加新项”命令，打开“添加新项”对话框，如图 2-5 所示。

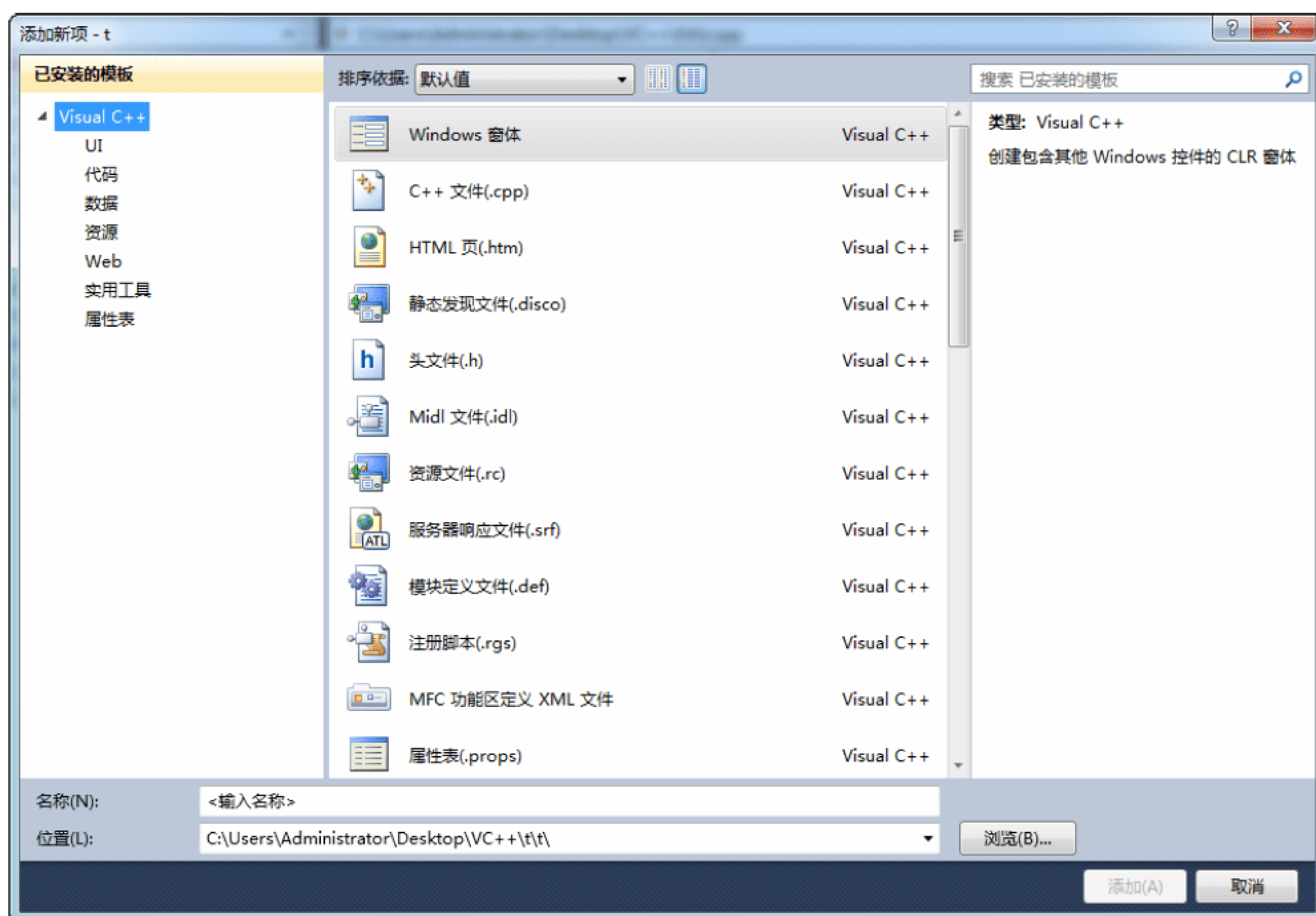


图 2-5　“添加新项”对话框

（2）选择“C++文件”类型，然后单击“添加”按钮，这样就成功地创建了新的源文件。

2.2.3　编译、链接程序

在 Visual Studio 2010 的 IDE 环境中，选择“生成”|“编译”命令或使用 Ctrl+F7 快捷

键，即可编译当前文件。编译过程会在输出对话框中显示编译结果，如图 2-6 所示。

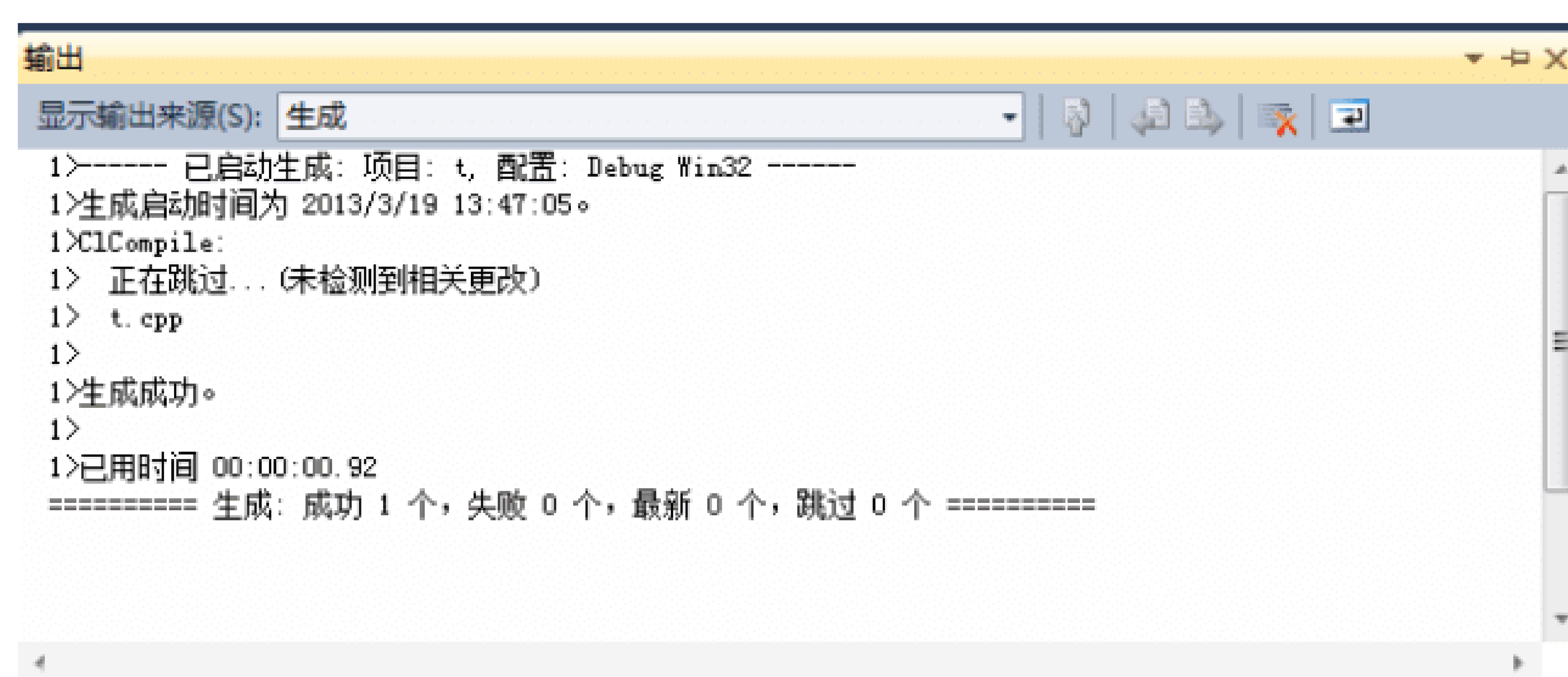


图 2-6　编译文件的结果

单个文件编译完成后，要生成可执行程序，就需要将多个编译后的文件链接在一起。

2.2.4　生成程序

链接程序完成后，就可以生成程序了。编辑、编译和链接，然后生成项目的过程如图 2-7 所示。

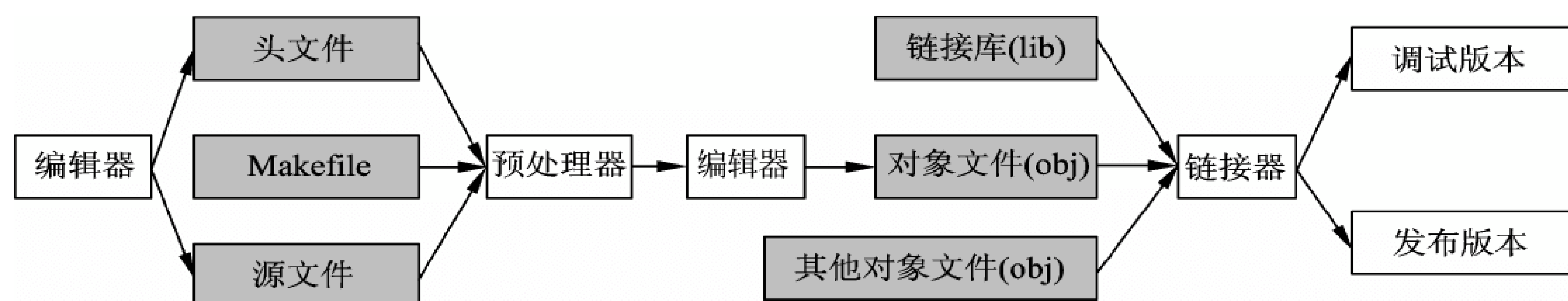


图 2-7　VC 生成程序的步骤

从图 2-7 中可以看出，程序员首先在编辑器中编辑源代码，Visual Studio 2010 中的源文件包括头文件和源文件。其中头文件存放函数和变量的声明，而源文件中存放程序的源代码。预处理器会处理头文件、源文件和 Makefile 编译文件，处理完后，交给编译器，编译器生成对象文件。接着，链接器会将生成的对象文件、其他用到的对象文件和链接库文件链接起来。最后，根据版本设置生成相应的版本。

生成项目的方法是，选择“生成”|“生成解决方案”命令或使用 F7 键，即可生成默认配置下的可执行文件。

在生成程序的过程中，会在“输出”对话框中显示生成信息，包括生成过程、警告信息、错误信息和成功信息。读者可以根据“输出”对话框中的信息确定生成的结果，如图 2-8 所示。

当生成过程中发生错误时，可以通过双击错误信息以定位到产生错误信息所在的源代码，并将光标移动到相应行上。

2.2.5　运行程序

运行程序也有两种方式：一种是在 IDE 环境中运行程序；另一种是在 Visual Studio 2010 环境外的 cmd 命令下运行程序。在 IDE 环境中运行程序的方法是，选择“调试”|“开始

执行（不调试）”命令或使用 Ctrl+F5 快捷键，运行结果如图 2-9 所示。在 cmd 命令下运行程序的步骤如下。

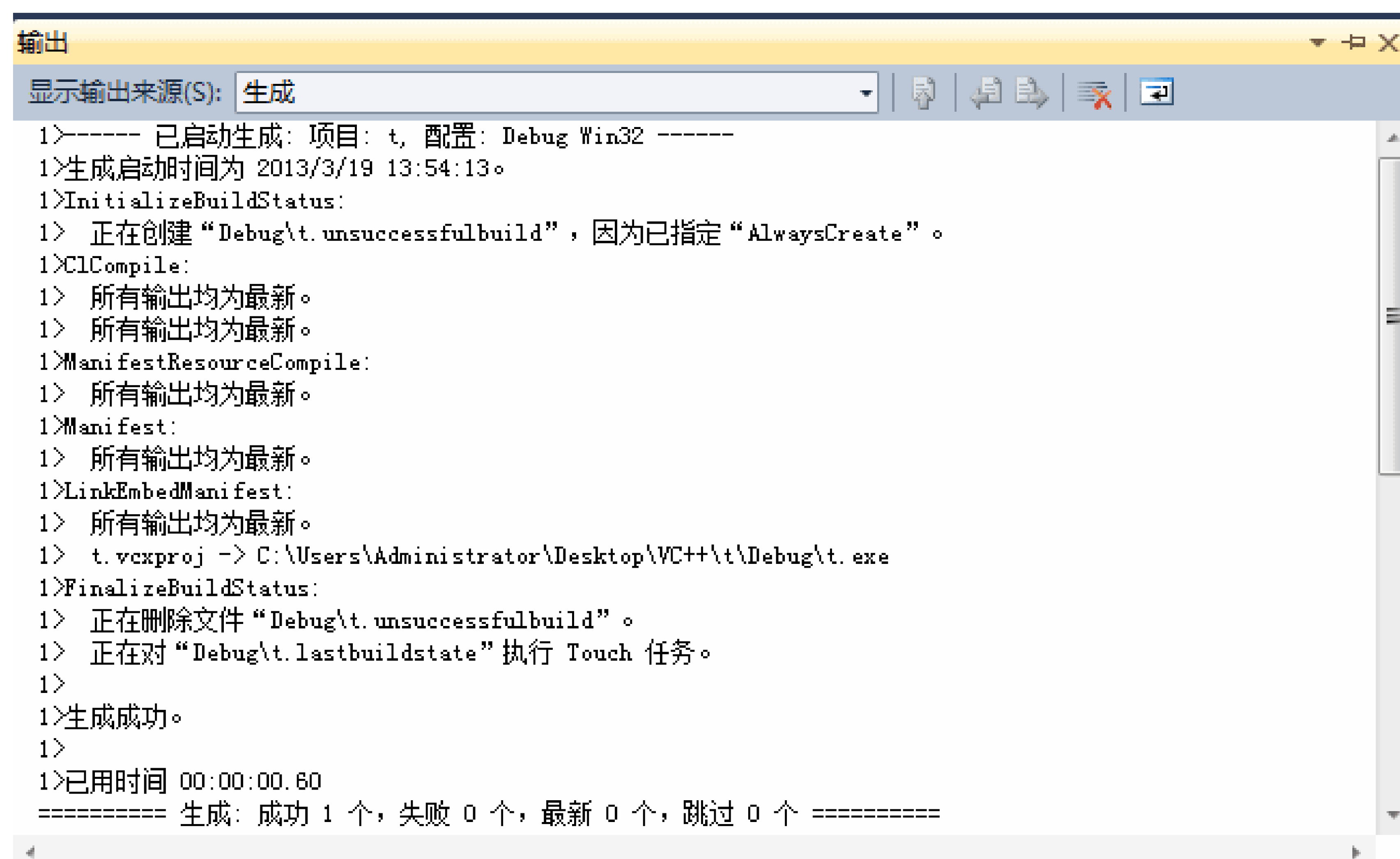


图 2-8　生成过程中的提示信息

（1）在 Windows 桌面下，选择“开始”|“运行”命令，打开“运行”对话框，如图 2-10 所示。

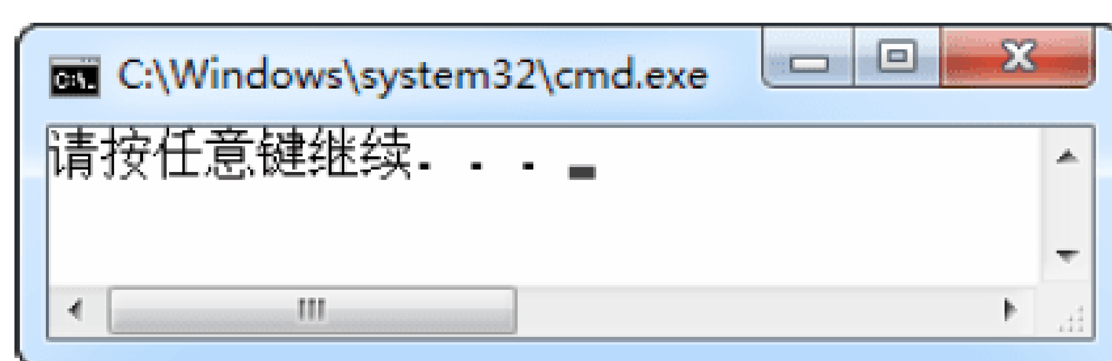


图 2-9　在 IDE 环境中的运行结果

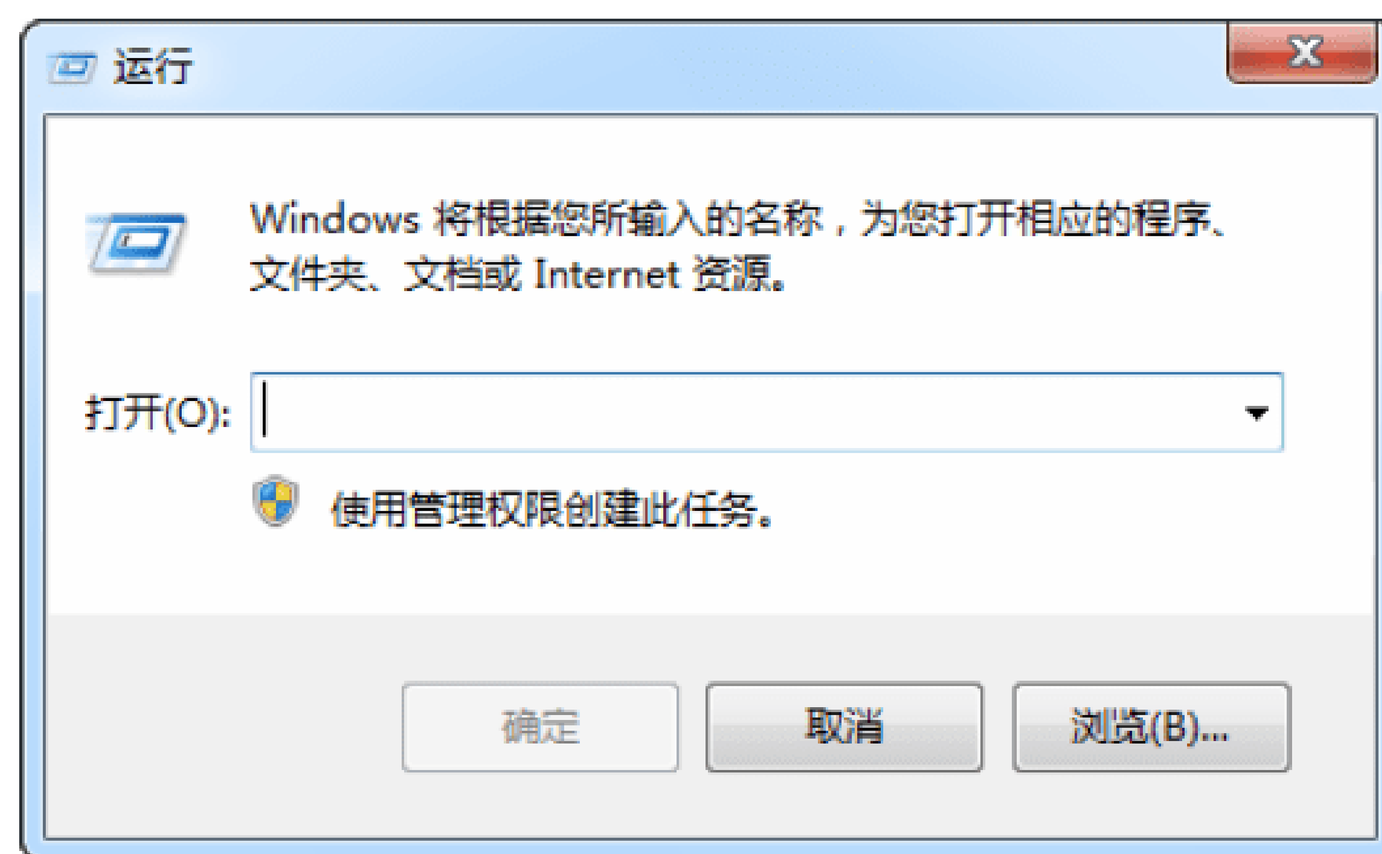


图 2-10　“运行”对话框

（2）在“打开”文本框中输入 cmd 命令，单击“确定”按钮，打开 Windows 的命令执行对话框，并更换当前路径为要运行的程序所在的路径。

（3）在命令提示符下，输入要运行的程序名称，按 Enter 键，运行结果如图 2-11 所示。



图 2-11　在 cmd 命令下运行

从图 2-11 中可以看出，两种方式的运行结果是相同的，即都没有输出（没在项目中添加输出代码），但程序确实运行了。唯一区别在于，在 Visual Studio 2010 中运行时，对话框中出现“请按任意键继续…”提示信息。要注意，此提示信息并不是运行结果的部分，

而是 Visual Studio 2010 为了停在结果部分而显示的提示信息。读者确认执行结果后，按下任意键盘键，程序即退出运行。

2.3　MFC 应用程序框架

为方便开发人员，微软提供了 MFC（Microsoft Foundation Class Library，微软基础类库），它封装了开发过程中常用的功能。其中，既包含基本的对字符串操作的 CString 类，也包括与架构有关的 CDialog 类和 CDocument 类等，在第 8 章会讲述常用的 MFC 类。MFC 应用程序是基于 MFC 的 Windows 平台下的可执行程序。本节将介绍 MFC 应用程序支持的框架及单文档结构程序。

2.3.1　创建 MFC 应用程序

Visual Studio 2010 提供了创建基于 MFC 应用程序的向导，使用 MFC 应用向导来引导用户进行一系列的设置。向导提供了基于 MFC 程序架构的文件的代码。它内建了部分功能，可以完成 Windows 中的部分基本操作。创建 MFC 应用程序的步骤如下。

使用项目模板向导创建项目，在项目列表中选择“MFC 应用程序”项目模板类型，单击“确定”按钮。进入如图 2-12 所示的“MFC 应用程序向导”页面。页面的正文部分是当前项目的设置信息。可以通过单击页面左侧的链接进入到相应的页面下进行设置，如图 2-13 所示为单击了“应用程序类型”链接。最后只要单击“完成”按钮，向导就会开始生成相应类型的框架程序。

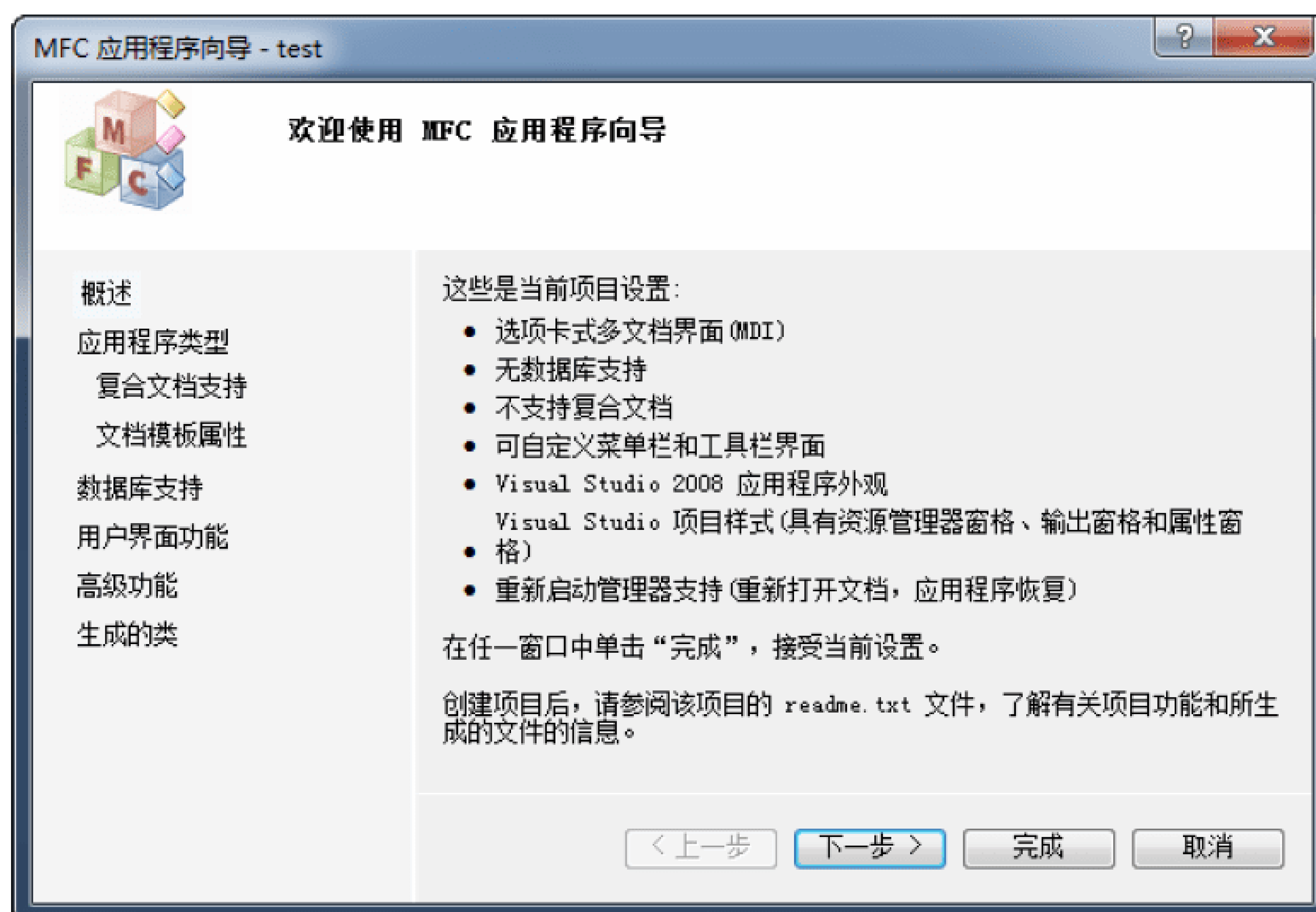


图 2-12　“MFC 应用程序向导”页面

使用 MFC 应用向导可以创建基于对话框的应用程序，也可以创建基于文档/视图的应用程序。

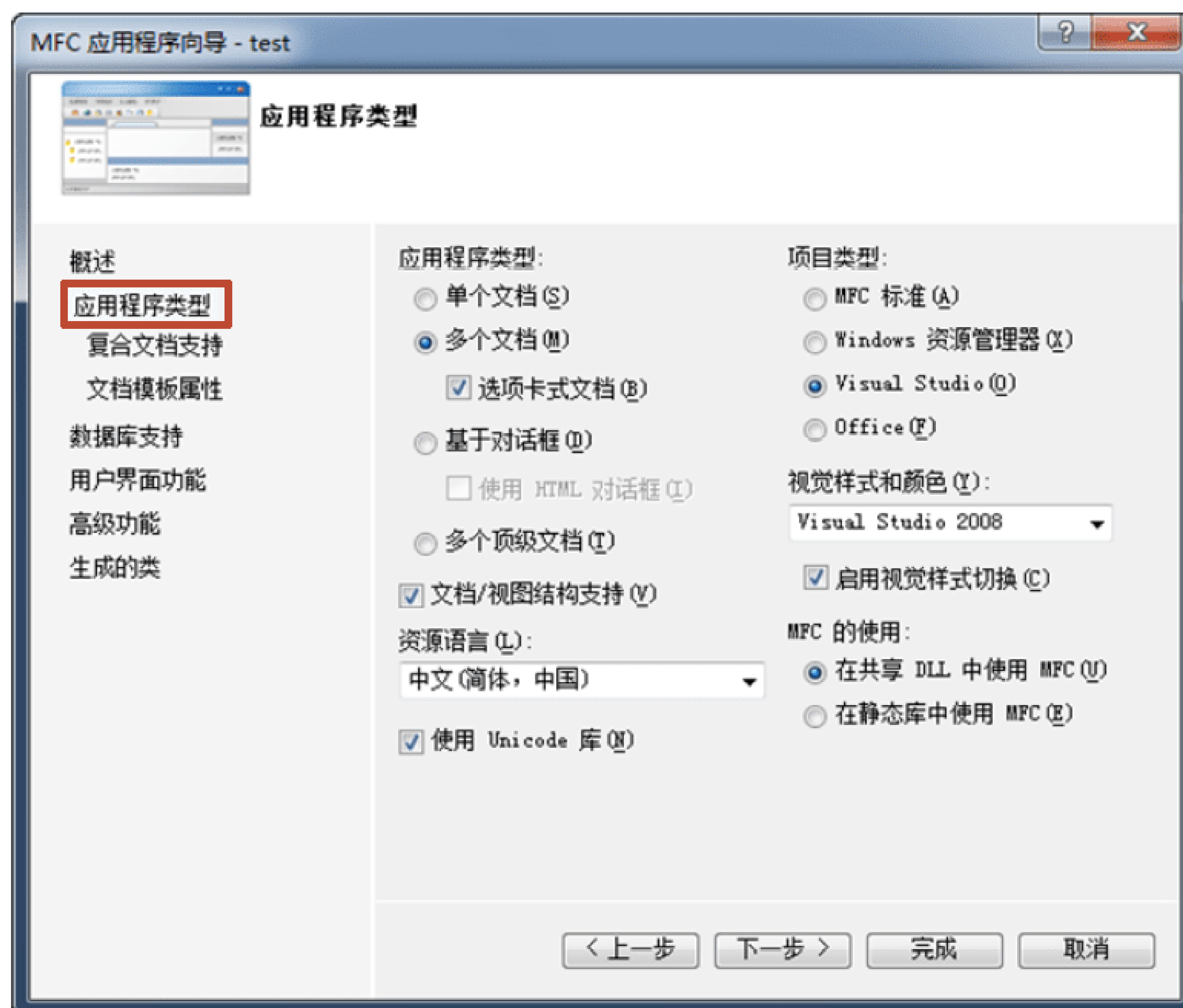


图 2-13　“应用程序类型”页面

2.3.2　认识文档/视图结构

MFC 框架提供了一种将程序数据的存储和显示分离的编程模型。在这个模型下，MFC 文档对象可以从存储器中读取和写入数据。文档还提供数据接口。分离的视图对象用于管理数据的显示，将数据显示在对话框中，供用户选择和编辑数据。视图从文档中获取显示数据，并将数据的修改通知文档。最常用的情况是，当用户需要单个文档的多个视图，如对同一组数据使用电子表格和图表视图，此时使用视图/架构框架非常合适。文档/视图模型允许单个视图对象代表数据的某个视图，而文档中的数据是所有视图共用的。当数据发生改变时，文档也会更新所有视图。

MFC 文档/视图架构支持多视图、多文档类型、分离对话框和其他用户接口属性。MFC 的文档/视图架构的核心主要由以下 4 个类实现。

- ❑ CDocument（或 COleDocument），用于存储和管理程序数据的对象。
- ❑ CView（或其派生类），用于显示文档数据和管理与数据交互的对象。
- ❑ CFrameWnd（或其派生类），管理单个文档和多个视图的对象。
- ❑ CDocTemplate（或 CSingleDocTemplate 或 CMultiDocTemplate），文档模板，它管理创建正确的文档、视图和框架对话框对象。

在 MFC 中，使用 CDocument 类提供程序员定义的文档类的基本功能。文档代表用户使用“文件”|“打开”命令打开、使用“文件”|“保存”命令保存的数据单元。使用 CView 类提供用户定义的视图类的基本功能。视图是关联到文档上的，并且充当文档和用户之间的中间对象。如视图可以在屏幕上显示文档的图片，并且在文档上执行用户的修改，同时视图也可以在打印机和打印预览上显示图片。图 2-14 表示了文档和视图之间的关系。

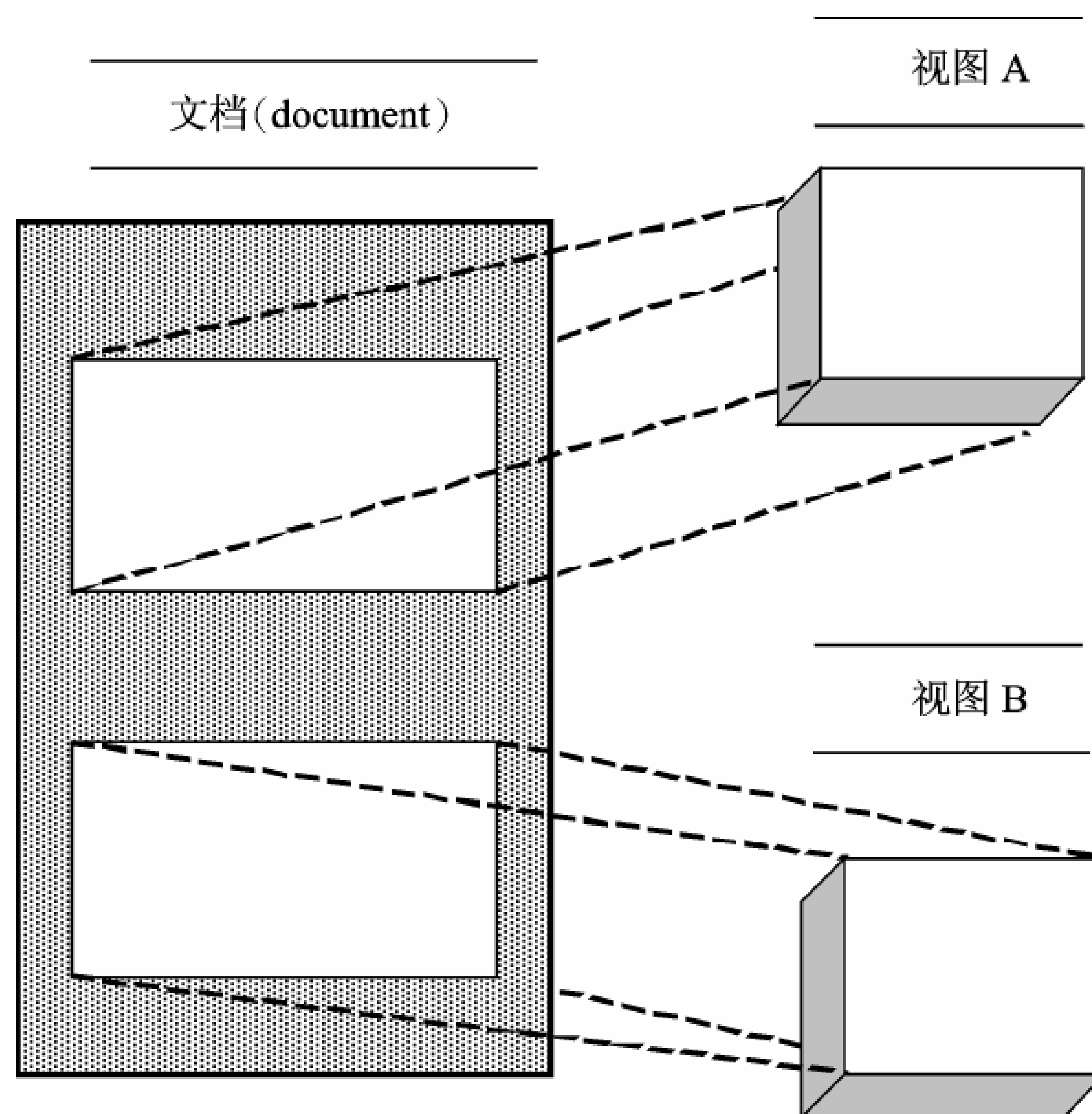


图 2-14　文档/视图关系图

2.4　本 章 小 结

本章介绍了在 Visual Studio 2010 中使用应用向导创建基本应用程序的步骤。本章的重点是掌握如何在 Visual Studio 2010 中创建 Win32 控制台应用程序和 MFC 应用程序。从第 3 章开始讲解 C/C++的语法。

2.5　习　　题

创建 Win 32 控制台应用程序，尝试添加代码，实现打印字符串“hello world”的功能。程序的运行效果如图 2-15 所示。

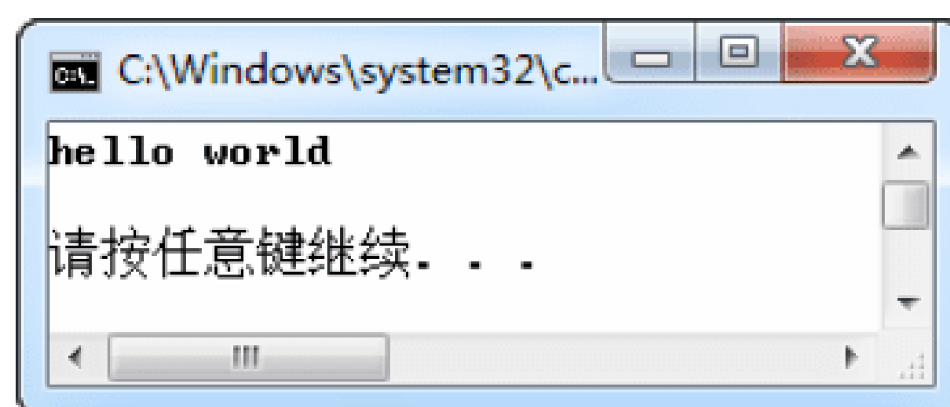


图 2-15　程序运行效果

【思路】参考 2.2 节的各小节，步骤依次为：创建控制台项目、新建源文件、添加代码以及编译运行程序。

第 3 章　C/C++语言基础

在第 1 章和第 2 章中介绍了 Visual Studio 2010 的开发环境和基本应用程序的创建。在 Visual Studio 2010 中创建了应用程序后，就需要了解 C++语言的语法和规则。只有深入了解语法规则及语法细节，才能开发出正确高效的程序。本章将详细讲述 Visual C++ 2010 的开发语言——C/C++的语言基础。

3.1　对标准 C 的扩展——C++

每种开发语言都有自己规定的结构和语法，只有编写的程序的结构和语法符合规定，相应的编译器才能正确处理。实质上，C 语言的编写就是数据定义和函数调用的组合。根据数据的特性，C 语言支持多种数据类型的定义，而对数据的操作则在函数调用中完成。程序入口是 main()函数，在 main()函数中调用其他功能函数。因此，C 语言是面向过程的开发语言。

C++是从 C 语言基础上发展而来的面向对象的编程语言，是对 C 语言的扩展，在保留了 C 语言的基本风貌的基础上，修正了 C 语言的弊端。C++语言主要在以下几个方面对 C 语言进行了扩展。

- C++语言的语法并不是全新的，这为原来的 C 语言开发人员从面向过程的开发语言过渡到面向对象的开发语言，提供了一个快速的转型过程。已有的 C 代码在 C++环境中仍然可以使用，只需要使用 C++编译器重新编译，并修正本来隐藏的错误就可以了。
- C++语言是更完善的 C 语言。C++语言是对 C 语言的扩展，不仅保留了良好的 C 语言习惯，并且修正了部分 C 语言的漏洞。如 C++语言对函数的声明做了强制规定，使得编译器可以检查函数的调用，减少错误发生的可能；C++语言加入了引用技术，使得函数调用者可以处理函数参数和返回的地址；C++语言引入了函数重载技术，使不同函数可以使用相同的函数名；C++语言引入了对命名空间的支持，扩大了函数的定义范围；并且提供了更完善的类型检查和编译时处理等。
- C++语言与 C 语言的运行效率基本一样。据不完全统计，相同条件下，使用 C++语言编写的面向对象的程序效率与 C 语言编写的程序相差在±10%左右。而且 C++语言的一些性能还可以调整程序的运行效率。
- C++语言是面向对象的，C 语言是面向过程的。因此，C++语言是用问题空间的概念描述问题的解决方法，而 C 语言是用解空间的概念描述问题的解决方法。所以，C++语言编写的程序比 C 语言编写的程序更容易理解。容易理解带来的好处就是

易于维护。通常维护工作是占用系统开销比较大的部分，因此 C++语言编写的程序的维护开销要比 C 语言编写的程序的维护开销要小。

- C++语言扩展了 C 语言对库的支持。使用库复用已有的代码可以大大提高开发效率，因此 C++语言也对 C 语言库的支持做了升级，它将库转换为类，当程序引入一个库，便向程序中引入一个新类，使得程序原有代码与引入的库浑然一体，风格一致，从而使得开发人员对库的使用更方便。
- C++语言引入了异常处理。这一点是对 C 语言的补充，因为 C 语言基本没有错误处理机制，C 程序对错误的处理，全靠开发人员自己实现。C++语言引入了异常处理，减少了开发人员对错误处理的程序的编写，并且增强了程序的健壮性。
- C++语言对复杂程序的支持比 C 语言要好。当程序非常复杂时，用于处理的变量和函数会非常多，比较容易发生命名冲突。因此，C++语言引入了命名空间机制，有了命名空间的限制，使用的变量和函数就可以无限制的增加。从而可以支持复杂程序的编写。据不完全统计，当 C 语言代码超过 50000 行时，命名冲突就成为问题，从而阻碍程序的开发。

C++语言由两种文件组成，即以.h 为扩展名的头文件和以.cpp 为扩展名的源文件，分别存放各元素的声明和数据、函数及类的定义。

3.2　C++语法元素

C++语法元素包括符号、注释、标识符、关键字、标点符号和操作符。本节同时还讲述了如何进行元素的声明和定义。

3.2.1　最小的元素——符号

C++符号是 C++程序中解析器可以识别的最小的元素。C++解析器可以识别多种符号，包括标识符、关键字、常数、操作符、标点和其他分隔符等。这些符号组合起来，就成为程序指令。符号被“空白”分隔开。空白可以是一个或多个下列元素的组合。

- 空格：当按下 Space 键时，输入的就是空格。
- 水平 Tab 键：此键根据系统定义，可以连续输入几个空格，一般是 4 个空格或 8 个空格。
- 换行：表示在编辑器中光标另起一行。
- 回车：当按下 Enter 键时，输入的就是回车。
- 注释：是用于描述代码的作用，方便开发人员标记程序的功能。

每个处理单元使用输入流处理，解析器使用从左到右的方向扫描输入流，创建更长的符号并从中分隔符号。例如代码如下：

```
a = i+++j;                    //自增一语句的使用示例
```

开发人员可能想实现下面两条语句中的一条：

```
a = i + (++j)
```

```
a = (i++) + j                    //编译器会按照此种方法解析上面的自增语句示例
```

因为解析器分析输入流时，使用从左到右的方向分析，所以，它会采用第二种解释方法。

3.2.2　注释规范

注释是写在程序代码中用于标记代码功能的符号，但是编译器在编译时，会将注释作为空格处理。虽然编译器在编译时忽略注释内容，但是它对程序开发来说非常重要，也是衡量程序质量的一个重要指标。注释的主要作用是注释代码，提供编写准确、适当的注释，对程序员和整个开发团队来说都非常重要，为后期维护和代码共享提供方便。C++支持两种注释方式——单行注释和块注释。

- 单行注释：以两个反斜杠开头，后面加注释内容。此注释方式表示//后一直到行尾的内容全部为注释。
- 块注释：以/*开始，以*/结束，其中的内容全部为注释。

下面代码说明了两种注释的使用：

```
int a=5;                              //定义整型变量 a，初始化为 5
/*定义整型变量 b，
初始化为 6*/
int b=6;
```

从上面的例子可以看出，在注释出现跨行时，最好使用块注释。当注释比较简短，一行足以显示时，使用单行注释比较简单。需要注意的是，注释是不支持嵌套的，例如：

```
/* 目的：注释整块代码
   问题：每行后的嵌套注释代码是无效的
   char a = 'A';                    /* 初始化字符 */
   cout << "a: " << a << "\n";      /* 打印字符 */
*/
```

上面代码是不能编译成功的，因为编译器在编译时，会为第一个/*查找与它匹配的第一个*/，即第一行的/*与第三行的*/匹配为一对。而第四行的/*与*/匹配为一对，第五行的*/没有匹配的注释符，因此，系统会提示编译错误。在使用单行注释要注意，不允许单行注释后跟行继续符，例如：

```
void main()
{
    printf( "This is a number %d",  //\
          5 );                      //此处使用单行注释会出现错误
}
```

上面的代码编译器进行编译时会提示错误，会将注释符后的行继续符下一行的内容作为空格进行编译，即“5);”会被忽略，因此，编译器会报语法错误。编译的代码如下所示，因此要注意单行注释后不要使用行继续符\。

```
void main()
{
```

```
    printf( "This is a number %d",
}
```

3.2.3　标识符命名规范

C++标识符，是系统预留的用于描述系统使用的元素的名称，由大小写的 26 个英文字母、0～9 之间的 10 个数字以及下划线组成，并且第一个元素必须是字母（大写或小写都可以）或者下划线。标识符是区别大小写的，如 hDevie 变量与 HDevice 变量是不同的。在 C++中下列元素需要使用标识符来表示。

- 对象或变量名：在内存中占据一部分空间，C++为它定义一个名称，在程序中使用对象名或变量名就可以直接访问存储空间中的值。如 int a;，语句中的 a 就是变量名。
- 类、结构或联合体名称：实质上是复杂类型的名称的标识符，用于标识不同种类的复杂类型。如 class Student 中的 Student 就是类名。
- 类型名称：表示简单类型的名称的标识符。如 int a 语句中的 int 为整型类型的标识符。
- 类、结构、联合体或枚举的成员：表示在类、结构、联合体或枚举中定义的成员变量的标识符。例如如果在 Student 类中定义 age 变量，则 age 就是类的成员标识符。
- 函数或类成员函数：表示函数名称的标识符。例如如果在 Student 类中定义 CheckIn()函数，则 CheckIn 就是类的成员函数的标识符。
- typedef 名称：表示类型重定义的标识符。
- 标签名称：表示 C++中用于标记 goto 语句可以跳转到的语句，此处主要用作语句指示。
- 宏名称和宏参数：使用#define 定义的宏的名称和参数。

在 C++中，不能使用关键字作为标识符。但是标识符中可以包含关键字。如 int 是一个非法的标识符，但是 pint 是合法的标识符。在 VC 中，标识符的最大长度为 247。

C++中在全局范围内预留以两个连续的下划线开头或者一个下划线后跟着一个大写字母的标识符，在文件范围内预留一个下划线后跟着一个小写字母的标识符。尽量不要使用这些形式的标识符，以避免与现在或将来预留的标识符冲突。

3.2.4　C++预定义的关键字

在 3.2.3 小节讲过，C++中不能使用关键字作为标识符，而实际上关键字是预定义的具有特殊意义的标识符。C++中预定义的关键字如表 3-1 所示。

表 3-1　C++关键字

asm	auto	bad_cast	bad_typeid
bool	break	case	catch
char	class	const	const_cast
continue	default	delete	do
double	dynamic_cast	else	enum

续表

except	explicit	extern	false
finally	float	for	friend
goto	if	inline	int
long	mutable	namespace	new
operator	private	protected	public
register	reinterpret_cast	return	short
signed	sizeof	static	static_cast
struct	switch	template	this
throw	true	try	type_info
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	while		

在 VC 中，以两个下划线开头的标识符是为编译器预留的。因此，在对 C++的实现中，在特定关键字前加两个下划线作为特定的 C++关键字。如表 3-2 是微软指定的 C++关键字。

表 3-2　微软指定的C++关键字

allocate	dllexport	__inline	__leave	property	__try
__asm	dllimport	__int8	__multiple_inheritance	selectany	uuid
__based	__except	__int16	naked	__single_inheritance	__uuidof
__cdecl	__fastcall	__int32	nothrow	__stdcall	__virtual_inheritance
__declspec	__finally	__int64		thread3	

其中，__asm 替换标准 C++中的 asm。而 allocat、dllexport、dllimport、naked、nothrow、property、selectany、thread 和 uuid 关键字是在使用__declspec 时才有效。默认情况下，VC 支持微软的 C++扩展，但是在编译时指定/Za 命令行选项（ANSI-Compatible）可以关闭此支持。当微软扩展打开时，用户可以在程序中使用上表中列出的关键字。在 ANSI 方式下，需要在这些关键字前加上双下划线标注。为了向后兼容，除了__except、__finally、__leave 和__try 关键字外，同时支持单下划线的关键字和 cdecl 关键字。

3.2.5　标点符号

C++中的标点符号是有语法的，并且对编译器来说，是具有语义的，但是标点符号本身没有语义。有些标点符号，不管是单独的还是组合的，也是 C++操作符或是对预处理器有语义。操作符主要有：

```
! % ^ & * ( ) - + = { } | ~ [ ] \ ; ' : " < > ? , . / #
```

其中，符号 [ ]、( )和 { }必须是成对出现的。

3.2.6　操作符

C++语言包括了所有 C 语言操作符，并且增加了一些 C++特有的操作符。C++操作符分为一元操作符、二元操作符和三元操作符。操作符在进行运算时，严格按照操作符优先

权的顺序进行运算，具有高优先权的运算符先于低优先权的运算符运算，操作符在进行操作时，有“方向性”。处在同一级别的操作符具有相同的优先级，此时，执行顺序为从左向右运算。通过小括号，可以改变操作符的运算顺序。表 3-3 列出了 C++支持的操作符。执行顺序是按照从高到低的优先权依次执行的。

表 3-3　C++操作符

操　作　符	名　　称	方　　向
::	范围确定符	无
::	全局符	无
[ ]	数组下标	从左向右
( )	函数调用	从左向右
( )	转换	无
.	对象的成员选择	从左向右
–>	指针的成员选择	从左向右
++	自增一后缀	无
– –	自减一后缀	无
new	分配对象	无
delete	删除对象	无
delete[ ]	删除对象	无
++	自增一前缀	无
– –	自减一前缀	无
*	乘	无
&	地址符	无
+	加	无
–	减	无
!	逻辑否	无
~	位与	无
sizeof	对象大小	无
sizeof ()	类型大小	无
typeid()	类型名称	无
(type)	类型转换	从右向左
const_cast	类型转换	无
dynamic_cast	类型转换	无
reinterpret_cast	类型转换	无
static_cast	类型转换	无
*	类成员的应用指针	从左向右
–>*	类成员的指针	从左向右
*	乘	从左向右
/	除	从左向右
%	取模	从左向右
+	加	从左向右
–	减	从左向右

续表

操　作　符	名　　称	方　　向
<<	左转换	从左向右
>>	右转换	从左向右
<	小于	从左向右
>	大于	从左向右
<=	小于等于	从左向右
>=	大于等于	从左向右
==	等于	从左向右
!=	不等于	从左向右
&	位与	从左向右
^	位异或	从左向右
|	位或	从左向右
&&	逻辑与	从左向右
||	逻辑或	从左向右
e1?e2:e3	条件	从右向左
=	赋值	从右向左
*=	乘赋值	从右向左
/=	除赋值	从右向左
%=	模赋值	从右向左
+=	加赋值	从右向左
－=	减赋值	从右向左
<<=	左转换赋值	从右向左
>>=	右转换赋值	从右向左
&=	位与赋值	从右向左
|=	位或赋值	从右向左
^=	位异或赋值	从右向左
,	逗号	从左向右

3.2.7　声明与定义

C++使用声明告诉编译器程序中定义了哪些程序元素或对象，而定义则说明了元素所执行的代码或数据。对象在使用前必须先声明。一条声明语句可以声明一个或多个对象。通常程序中需要多次使用声明。要使用多声明，则多声明中定义的元素类型必须相同。除了下面的情况，声明也可以作为定义来使用。

- 声明是一个函数原型，即声明函数，但是没有函数体。
- 包括 extern 标识符时，但是对象和变量没有初始化，或者函数没有函数体。此时表示在当前处理单元中不用进行定义，给定的是外部的连接名称。
- 在类声明中的静态数据成员。因为静态类数据成员是被类的所有对象共享的不连续的变量，因此，必须在类声明的外部定义和初始化。
- 没有定义的类名称的声明，如 class T。
- 使用 typedef 关键字声明的类型。

下面的代码给出了声明也作为定义的用法：

```
int i;                                  //声明和定义整型 int 变量 i
int j = 10;                             //声明和定义整型 int 变量 j
enum suits { Spades = 1, Clubs, Hearts, Diamonds };    //声明枚举类型
class CheckBox : public Control         //声明继承自 Control 的 CheckBox 类
{
public:
    Boolean IsChecked();                //判断是否选中的函数声明
    virtual int ChangeState() = 0;      //选择状态变量的函数声明
};
```

下面的代码给出了声明不作为定义的用法：

```
extern int i;                                          //声明外部整型变量 i
char *strchr( const char *Str, const char Target );    //声明 strchr()函数
```

定义是对象或变量、函数、类或枚举型的唯一说明。因为定义必须是唯一的，所以一个给定的程序元素只能包含一个定义。在声明和定义之间是多对一的关系，同一定义的元素，可以在程序多处声明。有如下两种情况，程序元素可以被声明但是不用定义。

- 函数声明了，但是未被其他函数调用或使用此函数地址的表达式。
- 类仅被使用，但是不需要知道其定义。这也是需要将声明放在.h 头文件中的原因。因此类必须声明。代码如下：

```
class WindowCounter;            //声明引用的外部类 WindowCounter，此类没有定义
class Window                    //声明类 Window
{
   static WindowCounter windowCounter;     //不需要 WindowCounter 定义
};
```

3.3　常量和变量

在编写程序的过程中，需要使用“一些可以区分的实体”存储开发过程中的值。这就产生了常量和变量。C++中使用“一串有意义的字符的组合”（即标识符）标识常量和变量，这样在程序设计中，就可以使用这些常量和变量存取需要的值。对于常量和变量的定义及使用 C++有其语法和规定。本节就介绍有关常量和变量的使用。

3.3.1　定义常量

常量顾名思义就是固定的量，在有效范围内值是不可以变化的。在 C++中使用 const 关键字定义常量，表示标识符表示的值是常量，告诉编译器，不允许程序代码修改它的值。语法如下：

```
//定义一个数据类型为 datatype，名称为 name，取值为 value 的常量
const datatype name=value;
```

上述语法表示定义一个数据类型为 datatype，名称为 name，取值为 value 的常量。如

下代码定义一个数据类型为整型，常量名为 fee 并且取值为 2 的常量：

```
const int fee=2;                  //定义一个整型的常量 fee，值为 2
```

在程序中，使用下列代码是错误的：

```
fee = 10;                         //错误：常量值是不可以修改的
fee ++;                           //错误：常量值是不可以修改的
```

C++中常量分为整型常数、字符常数、浮点常数和字符串常数 4 种类型。

1. 整型常数

整型常数表示没有小数部分或指数部分的数据常数，代表固定的整数。整型常数可以是十进制数（有效数字位为 0～9）、八进制数（有效数字位为 0～7）或十六进制数（有效数字位为 0～9，A～F）。其可以是有符号数，也可以是无符号数。可以是 long 类型，也可以是 short 类型的。可以是以 0 开头的数字代表八进制数，以 0x 或 0X 开头的数字代表十六进制数，以 u 或 U 结束的数字表示无符号数，以 l 或 L 结束的数字表示长整数，以 i64 结束的数字表示 64 位整数。如下代码显示了整型常量的定义方式。

```
const int a = 347;                      //十进制常量，值为 347
const int c = 0365;                     //八进制的 365
const int d= 0x55ff;                    //十六进制常量
const int e= 0X55FF;                    //十六进制常量，与 d 的值相等
const unsigned uVal = 256u;             //无符号数
const long lVal = 0x7FFFFEL;            //十六进制形式的长整型
const unsigned long ulVal = 076342ul;   //八进制形式的无符号长整型
```

2. 字符常数

C++字符常数是字符集的一个或多个成员，使用单引号引起来。VC 使用 ASCII 字符集。字符常数有 3 种形式——标准字符常数、多字符常数和宽字符常数。

```
const char ch = 'y';                    //标准字符常数
const int mbch = 'xy';                  //指定依赖于系统的多字符常数
const wchar_t wcch = L'xy';             //指定宽字符常数
```

此处 mbch 是一个 int 类型的。如果将它声明为 char，则其中的 y 将会被忽略。一个多字符常数可以包含 4 个字符，指定超过 4 个字符的字符常数时，会发生错误。因为一个字符是 8 位的，而 int 是 32 位的，因此，只能代表 4 个字符。

微软 C++支持标准、多字符和宽字符常数。使用宽字符常数可以指定扩展的可执行字符集的成员。标准字符常量使用类型 char，多字符常数使用类型 int，宽字符常数使用类型 wchar_t，这 3 种类型分别在 STDDEF.H、STDLIB.H 和 STRING.H 文件中定义，其中宽字符函数在 STDLIB.H 文件中定义。

标准字符常数和宽字符常数的定义方式的不同在于，需要在宽字符常数取值前加上字母 L。例如代码如下：

```
const char schar = 'a';                     //标准字符常数
const wchar_t wchar = L'\x81\x19';          //宽字符常数
```

从上面的代码中可以看出，在定义 wchar 宽字符常数时使用\x81 的形式，这里用到了转义符反斜线。在 C++中，使用反斜线加上指定码值代表特殊的字符。如表 3-4 所示列出了 C++中常用的转义字符。

表 3-4　C++常用转义字符

字　　符	ASCII 表示方法	ASCII 值	转 义 序 列
换行	NL (LF)	10 or 0x0a	\n
水平 Tab 键	HT	9	\t
垂直 Tab	VT	11 or 0x0b	\v
退格键	BS	8	\b
回车	CR	13 or 0x0d	\r
进制符	FF	12 or 0x0c	\f
反斜线	\	92 or 0x5c	\\
问号	?	63 or 0x3f	\?
单引号	'	39 or 0x27	\'
双引号	"	34 or 0x22	\"
八进制数	ooo	—	\ooo
十六进制数	hhh	—	\xhhh
Null 字符	NUL	0	\0

如果在反斜杠后的字符不是合法的换码字符，各种 C++处理各不相同，在 VC 中，会报 unrecognized character escape sequence 警告，表示不能识别的转义字符序列。

3．浮点型常数

浮点型常数用于表示包含小数部分的常数，默认类型为 double 型。浮点型常数包含小数点，也可以包含指数。浮点常数中的 e 或 E 后面部分的内容为指数部分，表示浮点数的数量级，数值为 10 的次幂，在常数前面的+或–表示符号，在常数后面的 f（或 F）、l（或 L）分别表示其类型为 float 或 long。下面是浮点型常数的例子。

```
const double = 18.46          //浮点型常数
const double = 38.            //浮点型常数
const double = 18.46e0        //浮点型常数，18.46 的 10 的 0 次幂
const double = 18.46e1        //浮点型常数，18.46 的 10 的 1 次幂
```

4．字符串常数

字符串常数是包含 0 个或多个字符集中的字符的字符串，使用双引号引起来。字符串常数是一个以 NULL 结束的字符序列。字符串的连接可以采用多种方式实现，例如代码如下：

```
char szStr[] = "12" "34";          //定义字符数组
char szStr[] = "1234";             //定义字符数组与上一行的作用相同
cout << "今天是星期一 "
       "并且是中秋节 "
       "也就是八月十五";            //输出内容，为一组字符串
```

```
cout << "今天是星期一  \
         并且是中秋节  \
         也就是八月十五.";                    //输出内容，为字符串
```

3.3.2　常量成员函数

除了常数类型的常量，还可以将类的成员函数定义为常量，即常量成员函数。在函数体内不可以修改任何数据成员，也不可以调用任何不是常量的成员函数。语法是在函数定义后加上 const 关键字，格式如下：

```
returntype class:functionname(param1,param2,……)  const
{
    //body—函数体
}
```

其中，returntype 表示返回类型，class 表示成员函数所属的类，functionname 表示成员函数的函数名称，param1,param2 等表示常量成员函数的参数，const 表示此成员函数是常量成员函数，{}中是成员函数的函数体。注意，在 C++中定义常量成员函数时，要在声明和定义后都加上 const 关键字，示例代码如下：

```
class Date
{
public:
   int getMonth() const;                   //获取当前月的取值的只读函数
   void setMonth( int mn );                //写函数，不能定义为 const
private:
   int month;                              //存放月的变量
};
int Date::getMonth() const                 //获取当前类中的月变量的值
{
   return month;                           //只是返回月变量的值，没有修改任何内容
}
void Date::setMonth( int mn )              //设置类中月变量的值
{
   month = mn;                             //修改数据成员
}
```

3.3.3　定义变量

与常量不同的是，变量在定义后，可以根据程序的需要对值进行修改。变量的定义方法与常量的定义方法类似，只是去掉了 const 关键字，其语法如下：

```
//定义一个数据类型为 datatype，名称为 name，取值为 value 的变量
datatype name=value;
```

以下代码定义了一个变量名称为 balance 的整型变量，并为其分配初始值 0。

```
int balance = 0;
```

3.3.4　代码的有效范围——作用域

每个 C++变量只能在程序的一定范围内使用，此范围称为变量的作用域。除了静态对象外，作用域可以确定变量的生命期。当调用类的构造函数和析构函数、变量在作用域内初始化时，作用域还确定变量的可见性。

1．作用域类型

C++中共分 5 种作用域，如下所述。

（1）本地作用域。在代码块中声明的变量只能在声明块中声明语句后访问。比如具有函数块中作用域的函数形参具有本地作用域，即在函数体内声明的变量只能在函数体内使用。例如代码如下：

```
{
    int i;                    //定义本地作用域的变量
}
```

上面代码中变量 i 在大括号内声明，因此，i 的作用域为本地作用域，只能在大括号内的代码中使用。

（2）函数作用域。只有标签属于此作用域的变量。标签可以在定义它的函数内任意地方使用，函数外部就不能使用。

（3）文件作用域。在代码块或类的外部声明的变量具有文件作用域。可以在处理单元中声明点后的任何地方访问。文件作用域对象既可以是静态对象也可以是非静态对象，其中具有文件作用域的非静态变量通常称为全局变量。

（4）类作用域。类的成员变量具有类作用域。只能通过使用对象的成员选择操作符（.或->）或类对象的指针成员操作符（*或->*）访问。而非静态类成员数据可以看作类对象的本地作用域。例如代码如下：

```
class Point             //定义代表点的类
{
    int x;              //点的 X 坐标，具有类作用域，也可以看作类对象的本地作用域
    int y;              //点的 Y 坐标，具有类作用域，也可以看作类对象的本地作用域
};
```

在上面的代码中，Point 类的 x 和 y 成员的作用域可以看作 Point 类的类作用域。

（5）原型作用域。函数原型中声明的变量只在原型声明范围内有效。以下代码声明了 strcpy()函数原型，其中变量 szDest 和 szSource 的作用域是在函数原型声明的范围内。

```
char *strcpy( char *szDest, const char *szSource );    //strcpy()函数原型
```

2．理解作用域

虽然作用域的概念是隐形的，但是对于它理解不透，会在程序中出现错误或隐藏的问题。下面对作用域的理解做些说明。

（1）变量的声明点为声明后和初始化前的程序点；枚举类型的声明点是定义了标识符，

但是还没有初始化前。例如代码如下：

```
double dVar = 7.0;          //定义 double 类型的变量，并初始化为 7.0
void main()
{
    double dVar = dVar;     //将全局变量的值赋值给本地作用域的变量
}
```

在上面代码中，声明点在初始化之前，则本地 dVar 应该初始化成全局变量 dVar 的值，即 7.0。枚举值的处理方式是相同的。例如代码如下：

```
//定义 4 个值分别为 1、2、3、4 的常量
const int Spades = 1, Clubs = 2, Hearts = 3, Diamonds = 4;
enum Suits                  //定义 Suits 枚举类型
{
    Spades = Spades,        //错误
    Clubs,                  //错误
    Hearts,                 //错误
    Diamonds                //错误
};
```

上面代码定义了常量 Spades、Clubs、Hearts 和 Diamonds，这些常量的作用域为全局作用域，因此在枚举值 Suits 中使用这些常量定义枚举值是错误的。因此，在编写代码时，即使作用域不相同，也应该尽量避免名称重复。在 C++中，提供了一种隐藏名称的方法。使用这个方法可以将变量的作用域限制在一定范围内。例如代码如下：

```
Test()              //Test()函数
{
        int i = 0;              //定义函数作用域的 int 类型的变量 i，并初始化值为 0
        cout << "i=" << i << "\n"   //输出 i 的值
        {
            int i = 7, j = 9;       //定义局部作用域的变量 i 和变量 j
            cout <<"i="<<i <<"\n" <<"j=" << j <<"\n";
                                    //输出局部作用域的 i 值和 j 值
        }
        cout <<"i = " <<i << "\n";  //输出函数作用域的 i 值
}
```

在上面代码中，程序在函数中使用了一对大括号，将代码包括起来，在大括号内的变量 i 的作用域为大括号内，其值不会影响大括号外的变量 i 的作用域，运行结果如下：

```
i = 0
i = 7
j = 9
i = 0
```

（2）当在文件中声明具有文件作用域的变量或函数与块中定义的变量或函数名称相同时，可以通过使用作用域确定操作符（::）访问文件作用域的名称。例如代码如下：

```
#include <iostream.h>
int i = 7;                                          //定义文件作用域的变量 i
void main()
{
   int i = 5;                                       //定义块作用域的变量 i
   cout << "块作用域 i 的值为: " << i << "\n";       //输出块作用域 i 的值
```

```
    cout << "文件块作用域 i 的值为: " << ::i << "\n";   //输出文件作用域 i 的值
}
```

上面代码运行的结果为：

```
块作用域 i 的值为:5
文件块作用域 i 的值为: 7
```

（3）在同一作用域中，当函数与变量的名称相同时，通过在名称前加上前缀 class 表示访问的是类对象。例如代码如下：

```
class Account                                      //在文件范围内声明类 Account
{
public:
    Account( double InitialBalance )  { balance = InitialBalance; }
    double GetBalance()  { return balance; }       //返回账户中的余额值
private:
    double balance;                                //存放账户余额的变量
};
double Account = 15.37;                            //隐藏类名 Account
void main()
{
    class Account Checking( Account );             //限定 Account 作为类名
    cout << "账户余额为: " << Checking.GetBalance() << "\n";
}
```

上面代码定义了类名为 Account 的类和变量名为 Account 的全局变量。在 main()函数的第一行中，定义了变量名为 Checking 的对象。通过在 Account 前加上 class 前缀，表示定义的是 Account 类的对象变量，而括号中的 Account 表示的是全局变量 Account，类型为 double 类型。下面代码显示了使用 class 关键字声明 Account 类型的指针的方法。

```
//定义 Account 指针的类变量
class Account *Checking = new class Account( Account );
```

3.4　数 据 类 型

C++中包含 3 种数据类型：基本数据类型、派生数据类型和 C++类。基本数据类型主要指内置在语言中的数据类型，如 int、char、float 等。派生数据类型指从基本数据类型派生而来的新类型。本节主要介绍 C++中的基本数据类型和派生数据类型。因为类是 C++中一个比较重要的概念，所以将在第 4 章中详细讲解 C++类。

3.4.1　基本数据类型

C++中基本数据类型指内置在语言中的数据类型，分为 3 类，分别是定点类型、浮点类型和空类型（void）。定点类型指在固定长度的存储空间内准确的存储一个数值；浮点类型指使用近似值表示一个数值，其有可能带有小数部分；空类型指空值，任何变量都不可以定义为 void，主要作用是表示函数不返回任何值或者无类型或任意类型的数据。表 3-5 列出了 C++中的基本数据类型以及 VC 中存储相应数据类型使用的长度。

表 3-5　C++基本数据类型

种类	数据类型	说　　明	长度
定点类型	char	char 类型表示字符型。在 VC 中，表示 ASCII 字符。char 类型分为 unsigned char 和 signed char 两种，分别表示无符号字符型和有符号字符型。默认情况下，char 类型是指 signed char 类型。实际上，在编译时，将字符型按照整型处理	1 字节
	short	short 类型（又称为 simply short 或者是 short int）是长度大于 char 类型，小于 int 类型的整型，又称为短整型。short 类型分为 signed short 和 unsigned short 两种，分别表示有符号短整型和无符号短整型。short 类型也就是 signed short 类型	2 字节
	int	int 类型是长度大于 short 类型，并且小于 long 类型的整型，int 类型分为 signed int 和 unsigned int，分别表示有符号整型和无符号整型。int 类型也就是 signed int 类型	4 字节
	__intn	定长整型。其中，n 是以比特为单位的整型长度。n 的取值可以是 8、16、32 和 64。也就是说可以使用它定义 8 位、16 位、32 位或者是 64 位的整型，而不需要使用 short、int、long 等类型。这样可以避免不同的 C++标准环境有可能为这些类型分配的存储空间大小不相同。根据 n 的取值，长度不相同，长度是 n/8 个字节。如 8 位整型，长度为 1 个字节	不定
	long	long 类型（或者是 long int）是长度大于 int 类型的整型。long 类型分为 signed long 和 unsigned long 两种，分别表示有符号长整型和无符号整型。long 类型也就是 signed long 类型	4 字节
浮点类型	float	float 是长度最小的浮点类型	4 字节
	double	double 是长度大于 float 类型，并且小于 long double 的浮点类型	8 字节
	long double	long double 是长度等于 double 类型的浮点类型	8 字节
空类型	void	void 表示空值	

表 3-5 列出的定点数中默认情况下，不带 signed 和 unsigned 标识的类型分别表示对应的有无符号类型，如 int 表示 signed int 类型。需要注意的是，当使用/J 编译选项时，默认的 char 类型为无符号字符型，即 unsigned char，其他类型的默认类型没有变化，仍然是有符号类型。

__intn 类型的数据类型与具有相同长度的数据类型是等同的。在 Visual Studio 2010 环境下，__int8 类型与 char 类型等同；__int16 类型与 short 类型等同；__int32 类型与 int 类型和 long 类型等同。

注意： C++中各种基本数据类型所占的存储空间根据 C++实现的不同而不同，表 3-5 第四列列出的基本数据类型长度是指在 Visual Studio 2010 下各种类型数据所占的存储空间。

3.4.2　数据类型的转换方式

C++中有多种数据类型，但在实际使用时，经常会遇到需要在不同数据类型之间转换

数据的情况，这时，就需要对数据进行类型转换。C++中进行数据类型转换的语法格式为：

```
数据类型转换表达式=(类型名称)转换表达式
```

其中，转换表达式表示要进行数据类型转换的表达式，而类型名称表示要将转换表达式转换成的类型。类型转换为对象提供了在适当情况下显式地将它转换成其他类型的方法。当完成数据转换后，编译器会将转换表达式作为类型名称指定的类型处理。数据转换可以在任何可度量数据类型之间进行转换，而且显式数据类型转换的规则与隐式数据类型转换的规则是相同的。具体参看如下代码：

```
void printTypeCast()                    //数据类型转换示例
{
    double x = 57.98;                   //定义 double 类型的变量
    cout << " x=" << x << "\n";         //输出 double 类型变量值
    int y = (int)x;                     //将 double 类型的变量值转换为 int 类型的值，
                                        //并存入 y
    cout << "y=" << y << "\n";          //输出 int 类型的 y 的变量值
}
```

上面代码将 double 类型的 x 转换成 int 类型，并存储在 y 中输出到界面上。从 double 型转换成 int 型时，会将 double 型数据的小数部分后的内容去掉。运行结果如下：

```
x=57.98
y=57
```

3.4.3　数组

派生数据类型又可以分为直接派生数据类型和组合派生数据类型。其中，直接派生数据类型指从基本数据类型直接派生而来的数据类型，包括数组、函数、指针和引用等。这些类型所指向的核心数据是基本数据类型，因此称为直接派生数据类型。组合派生数据类型指将基本数据类型组合而成的新的数据类型，其中不止包含一种基本数据类型，因此称为组合派生数据类型，包括类、结构体和共用体。

数组是包含指定数目的特定类型的变量或对象的集合。如一个由整型派生而来的数组，是一个整型数组。下面的代码定义了一个具有 10 个 int 变量的数组和一个具有 5 个 SampleClass 类对象的数组。

```
int         ArrayOfInt[10];             //定义具有 10 个 int 类型元素的数组
SampleClass aSampleClass[5];            //定义具有 5 个 SampleClass 类型元素的数组
```

在 C++中使用数组元素访问操作符（[]）访问数组元素。使用方法是在表达式后加上[]后，并加上表示数组对象元素在数组中的位置的下标。其语法为：

```
数组标识符[下标表达式]
```

其中，下标表达式表示要访问的元素在数组中的位置，它必须是可整型化的表达式。要注意的是，C++中下标值是基于 0 起始的，即数据中的第一个元素的下标为 0，第二个元素的下标为 1，依次类推。数组标识符表示要获取的数组元素所在数组的指针值。上面所说的都是一维数组，数组还可以是多维的。多维数组是一个具有数组元素的数组。因此，

三维数组也就可以看作是一个具有二维数组的数组。其语法格式为：

```
数组标识符[下标表达式 1] [下标表达式 2]...
```

下标表达式是按照从左向右的运算方向。首先计算最左边的下标“数组标识符[下标表达式 1]”，地址生成一个指针表达式。然后再向此指针表达式添加[下标表达式 2]，形成一个新的指针表达式。依此类推，直到最后一个下标表达式添加上。当运算完成后，如果最后得到的指针指向的数据不是数组类型时，就可以使用间接访问操作符（*）获取该值。例如代码如下：

```
int nYearsMonthsDays[20][12][16];              //三维数组的示例
```

上面代码定义了一个三维数组，表示 20 年间的任何一天。第一维表示 20 年中的一年，第二维表示指定年份中的某月，第三维表示指定年份的指定月份的某天。

3.4.4　结构体

C++中的结构体与类相同，区别在于结构体的所有成员数据和函数默认情况下都是具有公开权限的，并且默认是公开继承的。C++中结构体的定义语法如下：

```
struct struct_name        //结构体名称
{
    //body 结构体定义
};
```

其中，struct_name 表示定义的结构体的名称，并且在//body 的位置定义结构体的成员变量，示例如下：

```
struct MyDateTime         //日期时间结构体
{
    int year;             //年
    int month;            //月
    int day;              //日
    int hour;             //时
    int minute;           //分
    int second;           //秒
};
```

上面代码定义了表示日期时间的结构体，结构体的成员变量 year、month、day、hour、minute、second 分别用于存储日期时间的年、月、日、时、分、秒。

3.4.5　共用体

C++提供了一种共享内存的存储方法，称为共用体。它是在同一内存空间中可以定义多种数据元素，而在同一时间只能包含一种数据元素的类型，数据元素的类型可以是简单数据类型，也可以是数组或类等复杂数据类型。共用体的成员代表共用体包含的数据种类。共用体的存储空间是其成员列表中占用空间最大的成员的存储空间大小。其语法格式如下：

```
union 共用体名称 { 成员列表 } ;
```

其中，共用体名称指定了共用体类型的类型名。成员列表中可以定义共用体成员数据，也可以定义共用体成员函数。共用体成员数据可以是各种数据类型，但是不能定义为具有构造函数和析构函数的类、不能定义为具有重载赋值操作符的类、不能定义为静态数据成员。共用体的成员函数与类中的函数是相同的，可以是一般的成员函数，也可以是特殊函数，如构造函数和析构函数，但是都不能定义为虚函数。共用体不具有派生性和继承性。下面代码演示了如何使用共用体，工程名为 SampleUnion。

```
01  #include <stdio.h>
02  #include <string.h>
03  #include <stdlib.h>
04  #include <LIMITS.H>
05
06  enum NumType                          //声明一个枚举类型来描述要输出的类型
07  {
08      INTEGER_INT,                      //整型类型
09      INTEGER_LONG,                     //长整型类型
10      INTEGER_DOUBLE                    //double 类型
11  };
12  union NumValue                        //声明一个包含下面 3 种类型的共用体
13  {
14      int         iValue;               //int 类型值
15      long        lValue;               //long 类型值
16      double      dValue;               //double 类型值
17  };
18
19  void main( int argc, char *argv[] )
20  {
21      int count = argc - 1;                          //计算输入的参数个数
22      NumValue *Values = new NumValue[count];        //存放值的共用体
23      NumType *Types = new NumType[count];           //存放类型的数组
24      for( int i = 1; i < argc; ++i )                //循环处理每个参数
25      {
26          //判断输入参数中是否包含小数点
27          if( strchr( argv[i], '.' ) != 0 )
28          {
29              //为 dValue 成员赋值,并记录类型
30              Values[i].dValue = atof( argv[i] );
31              //记录数组的成员的类型为 double 型
32              Types[i] = INTEGER_DOUBLE;
33          }
34          else                                       //不是 floating 类型
35          {
36              if (( atol( argv[i] ) > INT_MAX )||(atol(argv[i]) < 0))
37              {
38                  //如果数据大于 int 类型的最大值，则将其存储在 lValue 成员中
39                  //并记录类型
40                  //将值转换成长整型
41                  Values[i].lValue = atol( argv[i] );
42                  //记录数组的成员的类型为长整型
43                  Types[i] = INTEGER_LONG;
44              }
45              else
46              {
47                  //否则，将其存储在 iValue 成员中,并记录类型
```

```
48                  Values[i].iValue = atoi( argv[i] );//将值转换成整型
49                  //记录数组的成员的类型为整型
50                  Types[i] = INTEGER_INT;
51              }
52          }
53          switch( Types[i] )          //根据类型种类，将种类信息和值信息输出
54          {
55          case INTEGER_INT:           //如果数据为整型，则输出整型值
56              printf( "数据类型为 Integer，值为%d\n", Values[i].iValue );
57              break;
58          case INTEGER_LONG:          //如果数据为长整型，则输出长整型值
59              printf( "数据类型为 Long，值为%d\n", Values[i].lValue );
60              break;
61          case INTEGER_DOUBLE:        //如果数据为 double 型，则输出 double 值
62              printf( "数据类型为 Double，值为%f\n", Values[i].dValue );
63              break;
64          }
65      }
66      system("pause");
67  }
```

在上面代码中，NumValue 共用体有 3 个成员，分别是 iValue、lValue 和 dValue，其分别是整型、长整型和双精度类型，根据具体为成员的赋值决定其类型。而其 3 个成员共享同一块内存。其内存分配如图 3-1 所示。

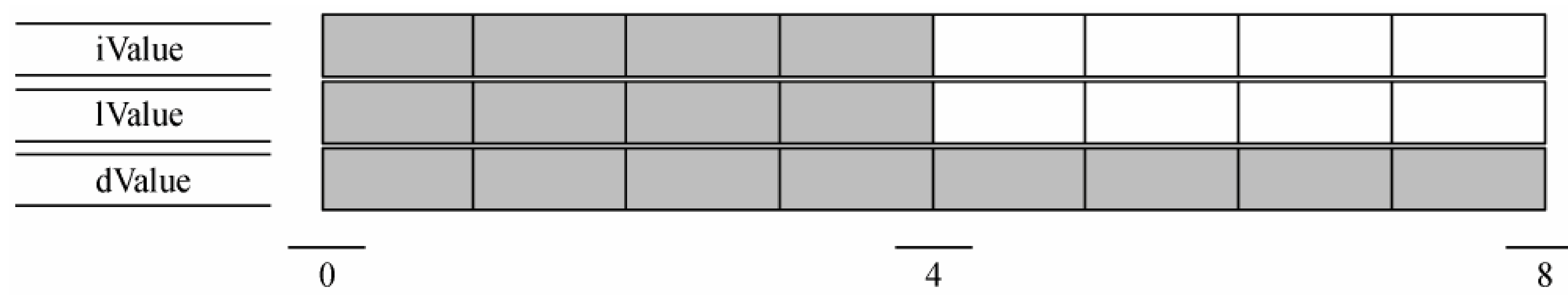


图 3-1　共用体的内存分配

上面代码首先取出通过命令行传入的参数个数，然后依次处理每个参数。在处理每个参数时，首先根据是否含有小数点判断输入的是否是浮点型。如果是浮点型，则将输入的参数值存入共用体的浮点型成员 dValue 中；如果不是浮点型，则根据转换后的值是否大于最大的整型值 INT_MAX 判断是整型还是长整型。根据结果，分别存入 iValue 和 lValue 成员变量中。最后根据判断的类型，输出类型信息和值。代码的运行结果如图 3-2 所示。

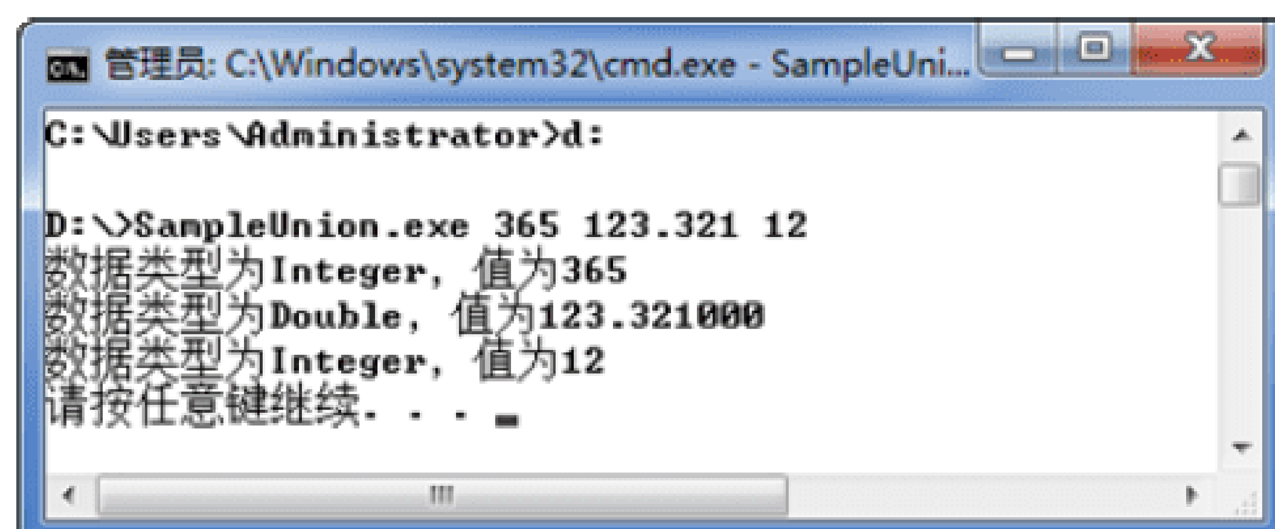


图 3-2　共用体示例的运行结果

3.4.6　匿名共用体

为了简化操作，C++还提供了一种特殊的共同体——匿名共用体，即没有声明共用体

名称的共用体。其语法格式为：

```
union { 成员列表 } ;
```

虽然匿名共用体与共用体一样，是成员共享同一块内存。但是匿名共用体不是声明类型，而是声明对象。因此，在匿名共用体中声明的名称不能与在其相同范围内声明的其他名称冲突。在匿名共用体中声明的名称可以直接使用，就像没有使用 union 声明的变量一样，只是其中的变量共享同一块内存。要使得共用体变成匿名共用体，除了要符合共用体的要求外，不能定义为静态，也不能包含成员函数。下面的代码用匿名共用体改写了上小节的程序。

```
01  #include <stdio.h>
02  #include <string.h>
03  #include <stdlib.h>
04  #include <LIMITS.H>
05
06  struct NumForm                          //表示数值的匿名共用体
07  {
08      enum NumType                        //声明一个枚举类型，用于描述要输出的类型
09      {
10          INTEGER_INT,                    //整型类型
11          INTEGER_LONG,                   //长整型类型
12          INTEGER_DOUBLE                  //double 类型
13      };
14      NumType type;                       //值的类型
15      union                               //声明一个包含下面 3 种类型的共用体
16      {
17          int         iValue;             //int 类型值
18          long        lValue;             //long 类型值
19          double      dValue;             //double 类型值
20      };
21      void print();                       //打印信息的函数
22  };
23  void NumForm::print()                   //根据数据类型，打印相应的信息
24  {
25      switch( type )                      //判断类型
26      {
27      case INTEGER_INT:                   //如果是整型，则输出整型值
28          printf( "数据类型为 Integer，其值为%d\n", iValue );
29          break;
30      case INTEGER_LONG:                  //如果是长整型，则输出长整型值
31          printf( "数据类型为 Long，其值为%d\n", lValue );
32          break;
33      case INTEGER_DOUBLE:                //如果是 double 型，则输出 double 值
34          printf( "数据类型为 Double，其值为%f\n", dValue );
35          break;
36      }
37  }
38
39  void main( int argc, char *argv[] )
40  {
41      int count = argc - 1;                           //计算输入的参数个数
42      NumForm *Values = new NumForm[count];           //存放输入的参数信息
43      for( int i = 1; i < argc; ++i )                 //循环处理每个参数
44      {
```

```
45          //floating 类型。为 dValue 成员赋值，并记录类型
46          if( strchr( argv[i], '.' ) != 0 )
47          {
48              Values[i].dValue = atof( argv[i] );//转换成 float 类型
49              Values[i].type = NumForm::NumType::INTEGER_DOUBLE;
50          }
51          else                                        //不是 floating 类型
52          {
53          {   //如果数据大于 int 类型的最大值
54              //则将其存储在 lValue 成员中，并记录类型
55              if (( atol( argv[i] ) > INT_MAX )||(atol( argv[i] )<0))
56              Values[i].lValue = atol( argv[i] );    //转换成 long 类型
57              Values[i].type = NumForm::NumType::INTEGER_LONG;
58              }
59              else
60              {   //否则，将其存储在 iValue 成员中，并记录类型
61                  Values[i].iValue = atoi( argv[i] );//转换成 int 类型
62                  Values[i].type = NumForm::NumType::INTEGER_INT;
63              }
64          }
65          Values[i].print();                                  //打印数值
66      }
67  }
```

从上面的代码中可以看出，在 NumForm 的 NumForm::print()成员函数中，对共用体的 3 个数据成员的访问就像声明数据成员一样，唯一区别就是共用体的 3 个数据成员共享同一块的内存。

3.4.7　枚举类型

枚举类型是用户子定义类型，包含一组命名的常数即枚举成员。默认情况下，第一个枚举成员的值为 0，每个连续的枚举成员比上一个枚举成员大 1，除非显式地为枚举成员指定值。枚举成员的取值可以重复。每个枚举成员的名称被作为一个常数，在 enum 定义的范围内必须唯一。枚举成员可以转换为整型值，但是，将整型值转换为枚举成员时需要显式转换，并且当枚举成员的取值重复时，结果是不确定的。C++中使用 enum 关键字指定枚举类型，语法格式为：

```
enum 枚举类型名称;                //定义枚举类型
```

在 C++中，在类中定义的枚举成员只能由类的成员函数访问，除非在它前面冠上类名（如，类名::枚举成员）。用户可以使用相同的语法直接访问类型名（即，类名::类型名）。如以下代码所示，使用 enum 关键字定义枚举类型。

```
01  enum Days
02  {
03      saturday,                   //saturday = 0 默认值
04      sunday = 0,                 //sunday = 0
05      monday,                     //monday = 1
06      tuesday,                    //tuesday = 2
07      wednesday,                  //依次类推
08      thursday,
09      friday
```

```
10  } today;                    //定义 Days 类型的变量 today
11  int  tuesday;               //错误，重复定义 tuesday
12  enum Days yesterday;        //在 C 和 C++中都合法
13  Days tomorrow;              //只在 C++中合法
14  yesterday = monday;         //为 yesterday 变量赋值为 Monday
15  int i = tuesday;            //有效，i = 2
16  yesterday = 0;              //错误，因为没有进行类型转换
17  yesterday = (Days)0;        //有效，但是结果不确定
18                              //不确定为 saturday 还是 sunday
```

上面代码演示了定义枚举类型 Days，其中存放每星期的星期数，并示范了如何使用枚举类型。

3.4.8　用 typedef 定义类型

C++不仅提供了丰富多样的类型，还提供了定义类型别名的关键字 typedef。typedef 可以为基本类型和派生类型定义别名。代码如下：

```
typedef unsigned char   BYTE;   //定义长度为 8 比特的无符号字符类型的别名为 BYTE
typedef BYTE *          PBYTE;  //定义 BYTE 指针的别名为 PBYTE
BYTE                    by;     //声明一个类型为 BYTE 的变量
PBYTE                   pbBy;   //声明一个指向 BYTE 类型的指针变量
```

由上面的代码可以看出，typedef 简化了数据类型的使用。对于比较长的数据类型，可以为其定义简短有意义的数据类型别名，这样方便使用。同时，当开发平台更换，根据需要对数据类型的使用发生变化时，只需要修改使用 typedef 的定义语句，就可以方便地实现数据类型的平台移植，而不需要修改程序中所有使用数据类型的地方。使用 typedef 不仅可以定义类型的别名，而且还可以定义函数类型的别名。代码如下：

```
void func1();                       //func1()函数原型
void func2();                       //func2()函数原型
typedef void (*PVFN)();             //定义 PVFN 为指向函数的指针，返回值为 void
PVFN pvfn[] = { func1, func2 };//声明函数指针数组
(*pvfn[1])();                       //执行函数指针数据中的第二个函数，即 func2()
```

上述代码定义了 func1()函数和 func2()函数，使用 typedef 关键字定义了函数类型的别名 PVFN，并定义了函数指针数组，初始化成员为 func1 和 func2，最后调用了函数指针数组中的第二个函数 func2()。

3.4.9　位域

类和结构可以包含占用比整型类型还小的存储空间的成员，这些数据成员称为位字段或位域。其语法格式如下：

```
数据成员声明 ： 占用的位数
```

其中，数据成员声明部分表示在程序中访问时使用的名称，必须是整型类型。占用的位数部分指定此成员在结构中占用的位数。代码如下：

```
struct Date
{
    unsigned nWeekDay  : 3;      //取值范围 0～7  （3 位）
    unsigned nMonthDay : 6;      //取值范围 0～31  （6 位）
    unsigned nMonth    : 5;      //取值范围 0～12  （5 位）
    unsigned nYear     : 8;      //取值范围 0～100 （8 位）
};
```

上面代码定义了时间结构，使用位域成员确定成员的存储。在位域定义中，如果定义的位数超过了其定义的类型的长度，则系统会自动分配新的存储单元作为后面的位域存储空间，而其类型与定义的类型是相同的。位域的存储空间的分配如图 3-3 所示。

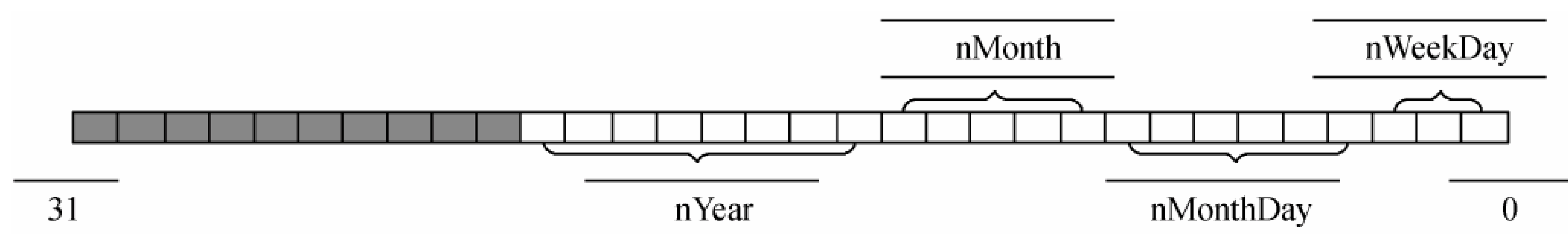


图 3-3　位域的内存分配

注意：在 Visual C++中，位域内存的分配是从低字节到高字节的。

如果忽略位字段的名称，则会填充数据剩下的位数，其后定义的成员会从新的存储空间开始重新分配。如下代码所示，声明了一个未命名的长度为 0 的字段：

```
struct Date
{
    unsigned nWeekDay  : 3;      //取值范围 0～7（3b）
    unsigned nMonthDay : 6;      //取值范围 0～31（6b）
    unsigned           : 0;      //强制对齐到下一个存储空间
    unsigned nMonth    : 5;      //取值范围 0～12（5b）
    unsigned nYear     : 8;      //取值范围 0～100（8b）
};
```

其中在 Date 结构中定义了一个长度为 0 的字段，这会使后面的字段从新的存储单元开始存储，如图 3-4 所示，nMonth 和 nYear 会从新的整型空间开始存储。

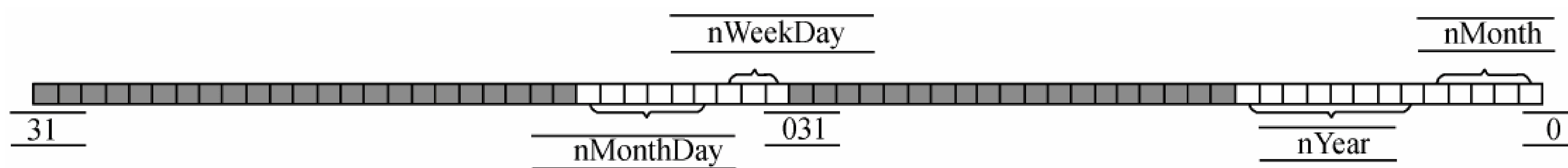


图 3-4　位域的扩展应用

位域可以充分利用存储空间，节约空间，提高程序运行效率。它在通信协议的解析等对存储位要求严格的情况下非常有用。但是在位域上进行操作时，不能操作位域的地址，也不能初始化位域的引用。

3.5　运算符和表达式

C++中变量是用来存储数据的，而运算符是用于操作数据的。使用运算符的语句组成

了表达式。C++语言中表达式的形式多种多样，但是其核心就是使用运算符操作变量或常量对象中的数据，完成预期功能。本节将介绍 C++中支持的运算符及其使用方法和注意事项。

3.5.1　算术运算符

C++提供了一组算术运算符，用于完成算术运算。其语法格式为：

```
左操作数 算术运算符 右操作数
```

上面的左操作数和右操作数是要进行算数运算的操作数，必须是可度量的数据类型。而算术运算符与现实世界中的算术运算的含义是相同的，如加法、减法、乘法、除法、取模等。系统支持的算术运算符如表 3-6 所示。

表 3-6　算术运算符

算术运算符	功　　能
+	加号运算符将两个操作数相加。两个操作数可以都是整型或浮点型，或者一个是指针型，一个是整型
–	减号运算符从第一操作数中减去第二个操作数。两个操作数可以都是整型或浮点型，或者一个是指针型，一个是整型
*	乘号运算符将两个操作数相乘。两个操作数可以都是整型或浮点型。使用乘号操作符时，得到的乘积会转换成结果代表的类型，但是它不处理溢出或下溢情况，如果存放结果的变量类型不够乘积结果的值，可能会丢失数据
/	除号运算符使用第一个操作数除以第二个操作数
%	取模运算符计算第一个操作数除以第二个操作数的余数

具体的使用方法，如以下代码所示。

```
void printMath()                              //算术运算符示例
{
    int a=7, b=8;                             //定义整型变量 a 和变量 b
    cout << "a+b=" << (a + b) << "\n";        //输出 a+b 的结果
    cout << "a-b=" << (a - b) << "\n";        //输出 a-b 的结果
    cout << "a*b=" << (a * b) << "\n";        //输出 a*b 的结果
    cout << "a/b=" << (a / b) << "\n";        //输出 a/b 的结果
    cout << "a%b=" << (a % b) << "\n";        //输出 a%b 的结果
}
```

上面代码依次演示了加法、减法、乘法、除法和取模运算符的使用。程序运行结果如下：

```
a+b=15
a-b=-1
a*b=56
a/b=0
a%b=7
```

3.5.2　赋值运算符

C++提供了一组赋值运算符，赋值运算符都是二元运算符，可以使用适当的赋值运算

符，执行运算后再赋值。使用赋值运算符的表达式称为赋值表达式。赋值运算符会将执行完运算的右操作数的值分配给运算符左边的表达式。因此，赋值操作的左操作数必须是可修改的。赋值后，赋值表达式具有左操作数的值。表 3-7 中列出了 C++支持的赋值运算符。

表 3-7　赋值运算符

赋值运算符	功　　能
=	简单赋值运算符，将右操作数赋值给左操作数
*=	乘赋值运算符，将右操作数和左操作数的乘积赋值给左操作数
/=	除赋值运算符，将左操作数除以右操作数的商赋值给左操作数
%=	取余赋值运算符，将左操作数除以右操作数的余数赋值给左操作数
+=	加赋值运算符，将左操作数和右操作数的和赋值给左操作数
–=	减赋值运算符，将左操作数减去右操作数的差赋值给左操作数
<<=	左移赋值运算符，将左操作数的二进制位向左移动右操作数个位，并赋值给左操作数
>>=	右移赋值运算符，将左操作数的二进制位向右移动右操作数个位，并赋值给左操作数
&=	位与赋值运算符，将左操作数和右操作数的与结果赋值给左操作数
|=	位或赋值运算符，将左操作数和右操作数的或结果赋值给左操作数
^=	位异或赋值运算符，将左操作数和右操作数的异或结果赋值给左操作数

具体的使用，参看如下代码：

```
void printAssignment()                                  //赋值运算符示例
{
    int a = 2, b = 3, c = 6, d = 9, e, f = 4;           //定义整型变量 a、b、c、d、
                                                        //e、f
    cout << "(e=d) e=" << (e=d, e) << "\n";             //输出(e=d, e)结果
    cout << "(a*=b) a=" << (a*=b, a) << "\n";           //输出(a*=b, a)结果
    cout << "(c/=b) c=" << (c/=b, c) << "\n";           //输出(c/=b, c)结果
    cout << "(a%=f) a=" << (a%=f, a) << "\n";           //输出(a%=f, a)结果
    cout << "(c+=d) c=" << (c+=d, c) << "\n";           //输出(c+=d, c)结果
    cout << "(c-=d) c=" << (c-=d, c) << "\n";           //输出(c-=d, c)结果
    cout << "(c<<=b) c=" << (c<<=b, c) << "\n";         //输出(c<<=b, c)结果
    cout << "(c>>=b) c=" << (c>>=b, c) << "\n";         //输出(c>>=b, c)结果
    cout << "(b&=d) b=" << (b&=d, b) << "\n";           //输出(b&=d, b)结果
    cout << "(c|=d) c=" << (c|=d, c) << "\n";           //输出(c|=d, c)结果
    cout << "(b^=c) b=" << (b^=c, b) << "\n";           //输出(b^=c, b)结果
}
```

上面代码依次演示了如何使用各种赋值运算符。其中还用到了逗号运算符，在后面会介绍。代码运行结果如下：

```
(e=d) e=9
(a*=b) a=6
(c/=b) c=2
(a%=f) a=2
(c+=d) c=11
(c-=d) c=2
(c<<=b) c=16
(c>>=b) c=2
(b&=d) b=1
(c|=d) c=11
(b^=c) b=10
```

3.5.3　关系运算符

C++提供了一组二进制关系和相等运算符，用于比较两个操作数之间的指定关系是否成立。如果关系成立，则关系表达式返回 1；否则，关系表达式返回 0。关系表达式的返回值类型为 int 类型。表 3-8 中列出了 C++支持的关系运算符。

表 3-8　关系运算符

关系运算符	功　　能
<	小于运算符。判断左操作数是否小于右操作数
<=	小于等于运算符。判断左操作数是否小于或等于右操作数
>	大于运算符。判断左操作数是否大于右操作数
>=	大于等于运算符。判断左操作数是否大于或等于右操作数
==	等于运算符。判断左操作数是否等于右操作数
!=	不等于运算符。判断左操作数是否不等于右操作数

下面代码显示了关系运算符的使用：

```
void printRelation()                                //关系运算符示例
{
    int a = 1, b = 2, c = 2, d = 4, e = 5;          //定义整型变量 a、b、c、d、e
    cout << "(a<b)=" << (a<b) << "\n";              //输出(a<b)结果
    cout << "(c<=b)=" << (c<=b) << "\n";            //输出(c<=b)结果
    cout << "(d>e)=" << (d>e) << "\n";              //输出(d>e)结果
    cout << "(e>=d)=" << (e>=d) << "\n";            //输出(e>=d)结果
    cout << "(b==c)=" << (b==c) << "\n";            //输出(b==c)结果
    cout << "(b!=c)=" << (b!=c) << "\n";            //输出(b!=c)结果
}
```

上面代码演示了关系运算符的使用，运行结果如下：

```
(a<b)=1
(c<=b)=1
(d>e)=0
(e>=d)=1
(b==c)=1
(b!=c)=0
```

在使用关系运算符时，要注意等于运算符“==”与赋值运算符“=”的使用，等于运算符是判断两个操作数是否相等，而赋值运算符是将右操作数的值赋值给左操作数。示例代码如下：

```
if (a ==b)        //判断 a 和 b 是否相等
{}
和
if (a = b)        //将 b 赋值给变量 a，并判断变量 a 的值是否大于 0
{}
```

虽然上面两条 if 语句都没有语法错误，但是含义是完全不同的。第一条表示判断变量 a 和变量 b 的值是否相等，第二条语句表示将变量 b 的值赋值给变量 a。有时会由于笔误，

将第一条语句写成第二条语句的形式，这样会导致程序的逻辑错误。

3.5.4 逻辑运算符

C++中有 3 种逻辑运算符，分别是逻辑与、逻辑或和逻辑非。除了逻辑非是一元运算符外，逻辑与和逻辑或都是二元运算符。逻辑与运算符符号为“&&”，用于运算两个操作数之间的与关系。当两个操作数都为非 0 时，则结果值为 1；否则，结果值为 0。运算方向为从左向右。代码如下：

```
void printLogicalAnd()                 //逻辑与运算符的示例
{
    int nCount=5, nPrice=2;            //定义数量和单价变量，并分别赋值为 5 和 2
    if ((nCount > 0)&&(nPrice > 0))
        //输出货物总价
        cout << "货物总价" << nCount*nPrice<< "元\n";
    else
        //如果数量或单价为 0，则提示输入的数据无效
        cout << "数据无效" << "\n";
}
```

上面代码使用逻辑与判断货物数量和货物单价的取值是否为有效范围。如果有效，计算货物总价；如果无效，则输出提示信息。运算结果为：

```
货物总价 10 元
```

逻辑或运算符符号为“||”，用于运算两个操作数之间的或关系。当两个操作数中的任何一个操作数为非 0 时，则结果值为 1；否则，结果值为 0。运算方向为从左向右。代码如下：

```
void printLogicalOr()                      //逻辑或运算符的示例
{
    int age=1000;                          //定义年龄变量，并初始化 1000
    if ((age > 120) || (age < 0))          //判断年龄值是否不在合理范围从 0～120
        cout << "年龄值无效" << "\n";
    else
        cout << "年龄值有效" << "\n";//如果年龄值有效，则提示年龄值有效
}
```

上面代码使用逻辑或判断年龄的取值是否为有效范围，此处定义年龄的有效取值范围为 0～120 岁，然后输出提示信息。运算结果为：

```
年龄值无效
```

逻辑非运算符符号为“!”，用于单个操作数的否运算。当操作数为 0 时，结果为 1；否则，结果值为 0。运算结果为 int 类型，此运算符的操作数必须为整数、浮点型或指针类型。代码如下：

```
void printLogicalNot()                         //逻辑非运算符的示例
{
    int balance=5;                             //定义余额变量，并初始化 5
    if (!balance)
```

```
        cout << "账户余额为 0 元" << "\n";      //判断余额是否为 0，并输出
    else
        cout << "账户余额不为 0 元" << "\n";    //余额不为 0，输出结果
}
```

上面代码使用逻辑非判断账户余额是否为 0，然后输出提示信息。运算结果为：

```
账户余额不为 0 元
```

3.5.5　位运算符

C++提供了对位进行操作的运算符，使用位运算符可以减少程序占用的空间，加快程序运行速度。位运算符主要有位与运算符、位或运算符、位异或运算符、位左移运算符、位右移运算符和反码运算符，如表 3-9 所示。

表 3-9　位运算符

位运算符	功　　能
&	位与运算符是二目运算符，比较第一个操作数的每位和第二个操作数相应的位。如果都为 1，则结果值相应的位也为 1；否则，结果值相应的位为 0
|	位或运算符是二目运算符，比较第一个操作数的每位和第二个操作数相应的位。如果其中有一个位为 1，则结果值相应的位也为 1；否则，结果值相应的位为 0
^	位异或运算符是二目运算符，比较第一个操作数的每位和第二个操作数相应的位。如果两个中一个为 0，另一个为 1，则结果值相应的位也为 1；否则，结果值相应的位为 0
<<	左移运算符是二目运算符，结果是将第一个操作数的各位向左移动第二个操作数指定的位数后的值。在移动过程中，原来左边多出的位数丢掉，右边的位数使用 0 补齐
>>	右移运算符是二目运算符，结果是将第一个操作数的各位向右移动第二个操作数指定的位数后的值。在移动过程中，原来右边多出的位数丢掉，左边的位数使用 0 补齐
~	二进制反码运算符是一目运算符，也称为补码或位非运算符，得出操作数的补码。操作数必须是整型类型的，并且运算后的结果值的类型与操作数相同。当操作数的某位为 1 时，结果值对应的位为 0；当操作数的某位为 0 时，结果值对应的位为 1

位运算符的使用，代码如下：

```
void printBitOperator()                        //位运算符示例
{
    int a=3, b=5, result;                      //0000 0011、0000 0101
    result = a & b;                            //位与运算符 0000 0001=1
    cout << "(a & b)=" << result << "\n";      //输出位与运算结果
    result = a | b;                            //位或运算符 0000 0111=7
    cout << "(a | b)=" << result << "\n";      //输出位或运算结果
    result = a ^ b;                            //异或运算符 0000 0101=6
    cout << "(a ^ b)=" << result << "\n";      //输出异或运算结果
    result = a << 2;                           //左移运算符 0000 1100 = 12
    cout << "(a << 2)=" << result << "\n";     //输出左移运算结果
    result = a >> 3;                           //右移运算符 0000 0000 = 0
    cout << "(a >> 3)=" << result << "\n";     //输出右移运算结果
    unsigned short c = 0xBBBB;                 //补码运算符 1011 1011 1011 1011
    c = ~c;                                    //0100 0100 0100 0100 = 17476
```

```
    cout << "(~c)=" << c << "\n";             //输出补码运算结果
}
```

代码运算结果如下：

```
(a & b)=1
(a | b)=7
(a ^ b)=6
(a << 2)=12
(a >> 3)=0
(~c)=17476
```

3.5.6　三目运算符

C++中只有一个三目运算符，即条件运算符（?：）。它是为了简化条件语句的编写而提供的运算符，其语法格式为：

```
逻辑表达式 ? 表达式 : 条件表达式
```

其中，逻辑表达式必须是整型、浮点型或指针类型，其后加上条件运算符，表示运算符会计算逻辑表达式的值是否为 true（即非 0）。如果是 true，则结果值为表达式代表的值；如果是 false（即 0），则结果值取条件表达式的值。这两个也称为结果表达式，加上冒号作为分隔符。无论是表达式还是条件表达式都可以是可计算的表达式，但是这两者不能同时是可计算的表达式。条件运算符的功能等同于如下 if 条件表达式：

```
if (逻辑表达式)
{
    结果 = 表达式;
}
else
{
    结果 =条件表达式;
}
```

如下代码显示了条件运算符的使用方法：

```
void printConditional()                                 //条件运算符
{
    int nBalance = 20, nAssign = 1, result;             //定义变量值
    result = (nBalance <= 0) ? 0 : nAssign ;            //判断余额是否为 0
    cout << "result=" << result << "\n";                //输出结果
}
```

上例使用三目条件运算符判断 **nBalance** 变量的值是否小于等于 0。如果是，则返回结果值为 0；否则，返回结果值为 **nAssign** 的值 1。上述代码等价于如下 if 条件语句：

```
if(nBalance <= 0)                     //判断 nBalance 变量是否小于等于 0
{
    result = 0;                       //赋值 result 为 0
}
else
{
    result = nAssign;                 //否则赋值 result 为 nAssign 变量的值
}
```

3.5.7　增 1 和减 1 运算符

增 1 运算符会将表达式的值增 1，减 1 运算符会将表达式的值减 1，分别是++和– –。其中，当增 1 或减 1 操作符出现在操作数前面时，称为前增 1 或前减 1 运算符，此时，会先将一元表达式的值增 1 或减 1 后，再使用该值。当增 1 或减 1 操作符出现在操作数后面时，称为后增 1 或后减 1 运算符，此时，会先使用表达式的值，然后将该值增 1 或减 1。后增 1 或后减 1 运算符的优先权比前增 1 或前减 1 运算符的优先权高。

```
++操作数                    //前增 1 运算符
--操作数                    //前减 1 运算符
操作数++                    //后增 1 运算符
操作数--                    //后减 1 运算符
```

无论是前增 1 或前减 1 运算符，还是后增 1 或后减 1 运算符，操作数必须是整型、浮点型或指针类型，并且是可以修改值的表达式。请参看下面的例子。

```
void printBeforeIncrement()           //前增 1 运算符的示例
{
    int a=1, b=2, result;             //定义变量 a、b 和 result
    result = (a) + (++b);             //result 现在的取值为 4
    cout << result << "\n";           //输出 result 值
}
void printBeforeDecrement()           //后增 1 运算符的示例
{
    int a=1, b=2, result;             //定义变量 a、b 和 result
    result = (a) + (b++);             //result 现在的取值为 3
    cout << result << "\n";           //输出 result 值
}
```

在上例中，printBeforeIncrement()函数中使用前增 1 运算符，会将 b 的取值增 1 变成 3 后再与 a 的值 1 相加，即结果为 4；而 printBeforeDecrement()函数中使用后增 1 运算符，会先将 b 的取值 2 和 a 的取值 1 相加后，再将 b 的取值增 1，即结果为 3。程序运行结果为：

```
4
3
```

3.5.8　逗号运算符

逗号运算符即连续赋值运算符，符号为逗号（,）。它是二元运算符，从左向右计算两个操作数。逗号运算符左边的操作数表示空表达式。运算的结果值与右操作符具有相同的类型。每个操作数可以是任何类型的数据。逗号运算符不能在操作数之间完成类型转换，它不能处理左值。在第一个操作数后有个顺序点，表示左操作数的所有求值会在右操作数求值开始之前完成。

逗号运算符通常用在上下文环境中只允许一个表达式时，计算两个或多个表达式。在有些情况下，它可以作为分隔符使用。所以要注意，逗号作为分隔符使用和作为逗号运算符使用是完全不同的。例如代码如下：

```
void TestFunction(int x, int y, int z)
{
    cout << "x=" << x << "\n";      //输出 x 值
    cout << "y=" << y << "\n";      //输出 y 值
    cout << "z=" << z << "\n";      //输出 z 值
}
void printComma()                   //打印运算结果值函数
{
    int a = 50, b = 0, c = 99;      //分别定义变量 a、b 和 c，这里使用逗号运算符
                                    //作为分隔符
    TestFunction(a, (b=47, b-7), c);    //输出 a、(b=47, b-7)和 c 的值
}
int main(int argc, char* argv[])    //程序主函数
{
    printComma();                   //执行 printComma()函数
    return 0;                       //返回
}
```

在上面代码中，TestFunction()函数具有 3 个参数。在 printComma()中调用 TestFunction()函数打印 3 个值。其中，第二个参数使用了逗号运算符，因此会顺序执行 b=47 语句和 b–7 语句，并将运算结果作为第二个参数传入。运算结果如下：

```
x=50
y=40
z=99
```

3.5.9　sizeof 运算符

sizeof 关键字用于计算表达式表示的变量或类型的存储字节数。此关键字返回一个 size_t 类型的值，计算的表达式是标识符或类型转换表达式。当计算结构类型或变量时，sizeof 返回实际大小，其中也包括为对齐而插入的字节。当计算静态数组的大小时，sizeof 返回整个数组的大小。sizeof 运算符不能返回动态分配的数组或外部数组的大小。例如代码如下：

```
struct align struct
{
   char ch;                             //字符型
   int i;                               //整型
};
void printSizeof()                      //sizeof 示例
{
    cout << "sizeof(int)=" << sizeof( int ) << "\n";
                                        //输出 int 类型的长度
    cout << "sizeof(align struct)=" << sizeof( align struct ) << "\n";
                                        //输出 align_struct 类型的长度
    int  array[] = { 11, 22, 33, 44 };  //定义整型数组
    cout << "sizeof( array )=" << sizeof( array ) << "\n";
                                        //输出数组长度
    cout << "sizeof( array[0] )=" << sizeof( array[0] ) << "\n";
                                        //输出数组元素长度
    cout << "count=" << sizeof( array ) / sizeof( array[0] ) << "\n";
                                        //输出数组个数
```

```
}
```

在上面代码中，首先输出 int 类型的大小，然后输出自定义结构 align_struct 的大小，接着输出定义的整个数组的大小和单个数组元素的大小，最后根据整个数组大小除以单个数组元素大小的方式计算数组个数，此方法是比较常用的计算数组包含的元素个数的方法。运行结果如下：

```
sizeof(int)=4
sizeof(align struct)=8
sizeof( array )=16
sizeof( array[0] )=4
count=4
```

注意：其中的 sizeof(align_struct)的运行结果与/Zp 选项有关，所以根据用户的设置不同，可能其结果值与此处不相同。

3.5.10　new 和 delete

new 关键字用于为指定类型的对象从空闲存储区中分配内存，并返回指向对象的对应类型的指针。如果失败，则 new 操作返回 0。用户可以通过编写自定义异常处理程序修改默认的操作，并将函数名作为参数调用运行时库函数_set_new_handler。其语法格式为：

```
[::] new [定位符] 类型名称 [初始值]
[::] new [定位符] (类型名称) [初始值]
```

如果重写对象的 new 方法，则定位符可以用于传递外加的参数。类型名称指定要分配的空间存储的变量的类型。如果类型是复杂类型，则可以通过使用括号强制绑定顺序。初始值用于提供初始化对象使用的值。不能为数组指定初始值，只有当类具有默认的构造函数时，new 操作符才可以创建数组对象。请参看下面的代码：

```
int *pint = new int;                     //创建指向 int 类型的指针
char *pchar = new char( 'X' );           //创建指向 char 类型的指针
Date *pdate = new Date( 2, 18, 2013 );   //创建指向 Date 类型的指针
char *pstr = new char[sizeof( str )];    //创建指向字符数组的类型的指针
char (*pchar)[10] = new char[x][10];     //创建指向二维字符数组的类型的指针
```

在上面代码中，第一条语句为整型分配了内存。第二条语句为字符类型分配了存储空间，并初始化为字符 X。第三条语句为日期类型分配了存储空间，使用类的带有 3 个参数的构造函数为其赋值为 2013 年 2 月 18 日，并将结果返回给日期类型的指针。第四条语句为字符数组分配存储空间，并将其指针分配给一个字符指针。第五条语句为大小为 x*10 的二维数组分配存储空间。当为多维数组分配空间时，除了第一维的维数外，其他维数必须为常数表达式。

如果使用没有任何外部参数的运算符，当构造函数抛出异常错误时，编译器会生成调用 delete 操作符的代码。如果使用占位符格式的 new 运算符或使用带外部参数的 new 运算符，当构造函数抛出异常错误，编译器不会生成调用 delete 操作符的代码。因此，此种情况下的异常发生后的内存释放工作应该由开发人员人工处理，否则就会出现 C++的典型程

序漏洞——内存泄露。代码如下：

```
MyClass* p1 = new MyClass(99);      //p1 指向的堆栈内存会被 delete 释放
//此调用，因为使用了带定位符的构造函数，因此，会产生内存泄露
MyClass* p2 = new(__FILE__, __LINE__) MyClass(99); //创建 MyClass 类
```

使用 delete 运算符可以释放由 new 运算符分配的内存空间。其语法格式为：

```
[::] delete 指针
[::] delete [ ] 指针
```

delete 关键字释放内存块。上面的指针参数必须是先前由 new 操作符分配的内存块的指针。如果指针指向数组，则当调用 delete 时，需要在指针前加上一对空中括号。代码如下：

```
delete pint;          //删除指向整型的指针
delete pchar;         //删除指向字符型的指针
delete pdate;         //删除指向日期型的指针
delete pstr[];        //删除指向字符串型的指针
delete pchar[];       //删除指向二维字符数组型的指针
```

上面代码依次释放前面申请分配的存储空间。

3.5.11　范围确定符

在 C++中，在变量名前加上范围确定符“::”前缀，即两个冒号，可以告诉编译器使用全局变量而不使用本地变量。即使代码上下文环境是嵌套的本地作用域，范围确定符也不能提供访问下个外层范围的变量，只能提供对全局变量的访问。示例代码如下：

```
#include <iostream.h>                               //范围确定符的示例
int pages = 800;                                    //全局变量
void printPages()
{
    int pages = 100;                                //本地变量
    cout << "全局变量 pages=" << ::pages<< '\n'; //打印全局变量
    cout << "本地变量 pages=" << pages<< '\n';   //打印本地变量
}
int main()                                          //程序主函数
{
    printPages();                                   //调用 printPages()函数
    return 0;                                       //返回
}
```

在上面的例子中，有两个名为 pages 的变量。第一个是全局变量，值为 800 页。第二个为 printPages()函数的本地变量。两个冒号告诉编译器使用全局 pages 变量而不是本地 pages 变量。运行结果如下：

```
全局变量 pages=800
本地变量 pages=100
```

3.5.12　类成员访问符

C++提供访问结构体、联合体和类的成员的操作符（.和->），其语法格式为：

```
表达式.成员选择标识符            //第一种方式
表达式->成员选择标识符           //第二种方式
```

其中，表达式表示类型为结构体、联合体或类等复合对象的变量或指针。第一种方式中，表达式代表的是对象变量；第二种方式中，表达式代表的是指针。成员选择标识符表示表达式类型的成员函数名称。操作的结果值是标识符的返回值。

实际上，如果表达式在前面包含间接访问操作符“*”应用的指针值，使用“->”成员选择符的表达式是使用“.”成员选择符的表达式的简写版本。因此，当表达式是指针类型时，下面两种成员选择表达式的作用是相同的：

```
表达式->成员选择标识符              //指针成员访问符
(*表达式).成员选择标识符            //对象成员访问符
```

具体使用代码如下：

```
struct MyDate {                      //成员选择操作符示例
   int  nYear;                       //年
   int  nMonth;                      //月
   int  nDay;                        //日
};
void printMemberSelect()             //打印选择的成员函数
{
    MyDate tmpDates;                 //定义 MyDate 结构的变量
    tmpDates.nYear = 2008;           //为对象的 nYear 变量赋值
    cout << "年=" << (&tmpDates)->nYear << "\n";    //输出 nYear 变量的值
}
```

在上面的代码中，tmpDates.nYear 与(&tmpDates)->nYear 表达的含义是完全相同的，区别在于一个是用于赋值，一个是用于取值。其运行结果如下：

```
年=2008
```

3.5.13　成员指针操作符

C++提供成员指针操作符.和->来访问类的成员指针。这两个操作符都是二元操作符，组合第一个操作数和第二个操作数访问类成员，其语法格式为：

```
类表达式. 成员表达式                //第一种情况
类表达式-> 成员表达式               //第二种情况
```

- 第一种情况：类表达式必须是类类型的对象，成员表达式必须是成员指针类型。
- 第二种情况：类表达式必须是指向类类型对象的指针，成员表达式必须是成员指针类型。

下面是使用这两种成员操作符的示例，代码如下：

```
void printClassAccess()                    //成员访问符示例
{
    CMyDate myDate1;                       //定义 CMyDate 类型的变量
    myDate1.nDay = 17;                     //为 myDate1 变量的 nDay 成员赋值为 17
    CMyDate* myDate2 = new CMyDate();      //定义 CMyDate 类型的指针变量
    myDate2->nDay = 18;                    //为 myDate2 变量的 nDay 成员赋值为 18
    //输出变量的值
    cout << "myDate1.nDay=" << myDate1.nDay << ";myDate2->nDay="
    << myDate2->nDay << ".";
}
```

3.6 控制语句

所有的编程语言都是通过语句执行指令，而现实世界中语句间存在 3 种逻辑结构：选择结构、循环结构和跳转结构。所以，在 C++语言中提供了表达式语句、选择语句、循环语句和跳转语句。本节将介绍 C++中有关控制语句的语法。

3.6.1 表达式语句、空语句和复合语句

表达式语句可以计算表达式的值，结果不会控制程序的跳转，也不会重复执行。在表达式语句中的所有表达式都会被计算，并且在执行完下条语句前会完成计算。常用的表达式有赋值语句和函数调用。如下代码所示，其中两条语句都是表达式语句。

```
int z = 2*3 + 5;
int a = min(x,y);
```

C++也支持空语句，就是一个没有任何表达式的表达式语句，由一个分号构成，常用于在重复语句中作占位符，或是在复合语句或函数的结尾处放置标签的语句。下面的代码显示了如何使用空语句。

```
char *strcpy( char *Dest, const char *Source ) //复制字符串函数
{
    char *DestStart = Dest;                //定义字符指针，指向 Dest
    while( *Dest++ = *Source++ )           //将数据从源字符串中复制到目的字符串中，
                                           //直到到达字符串尾
   ;                                       //空语句
   return DestStart;                       //返回结果字符指针
}
```

复合语句也称为代码段（程序块），由放在一对大括号中的 0 个或多个语句组成。复合语句可以用在任何可以使用单条语句的地方。在使用复合语句时，要注意在复合语句中的声明语句，其作用范围仅在定义的复合语句中有效。例如，if 语句后的一对大括号中包含的就是复合语句，其中的变量 a 仅在 if 语句的复合语句中有效。例如代码如下：

```
if( Amount > 100 )                 //判断 Amount 变量的值是否大于 100
{
    int a = 30;                    //定义整型变量 a 的值为 30
    cout << a;                     //输出 a 的值
```

```
}
else
    Balance -= Amount;             //在余额中减去 Amount 的值
```

3.6.2　选择语句

选择语句提供了有条件地执行代码段的方法。C++中提供了两种选择语句：if 语句和 switch 语句。

1. if语句

if 语句根据求得的括号内表达式的值确定执行的代码块。其语法格式为：

```
if ( 表达式 )  语句 1
if ( 表达式 )  语句 1  else 语句 2
```

其中，表达式必须是算术类型或指针类型，或者是明确定义了转换成算术或指针类型的方法的类。当表达式的计算结果为非 0 值，即 true 时，执行语句 1 代码段；否则，跳过或执行语句 2 代码段，如图 3-5 所示。

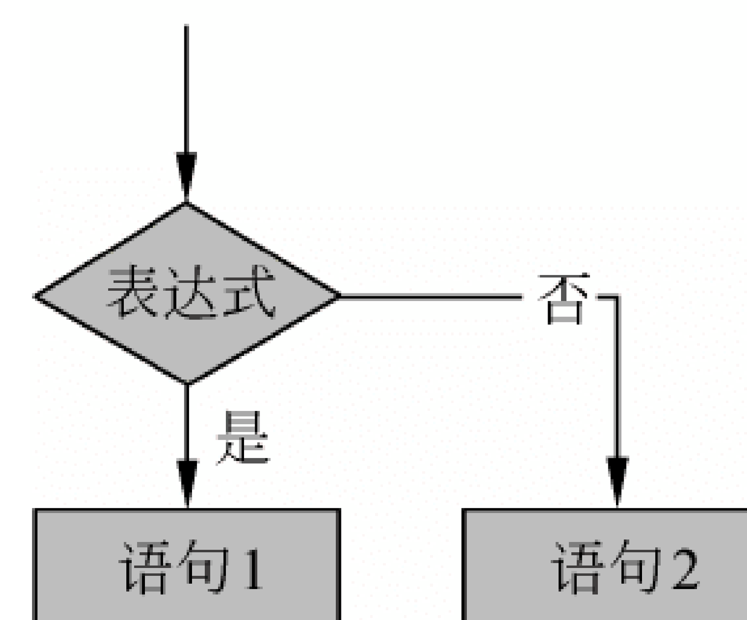


图 3-5　if 语句的执行流程图

当 if 语句中出现嵌套时，if…else 语句中的 else 语句与前面最近的还没有相应 else 语句的 if 语句进行匹配。如下代码显示了 if 语句的嵌套情况。

```
if( condition1 == true )
    if( condition2 == true )
        cout << "条件 1 为真; 条件 2 为真\n";
    else
        cout << "条件 1 为真; 条件 2 为假\n";
else
    cout << "条件 1 为假\n";
```

虽然，使用 if…else 语句的嵌套规则可以确定配对情况，但是建议在编写程序时，养成良好的编程习惯，使用大括号{}将 if 和 else 子句括起来，这样，可以清晰地显示 if 条件语句的逻辑关系。以下是将上面的代码用大括号改写后的代码。

```
if( condition1 == true )                  //如果条件 1 为 true
{
    if( condition2 == true )              //如果条件 2 为 true
    {
        cout << "条件 1 为真; 条件 2 为真\n";
    }
    else                                  //如果条件 2 不为 true
    {
        cout << "条件 1 为真; 条件 2 为假\n ";
    }
}
else                                      //如果条件 1 不为 true
{
    cout << "条件 1 为假\n";
}
```

2．switch语句

switch 语句根据表达式的取值在多个代码段中选择要执行的代码段。语句流程如图 3-6 所示。

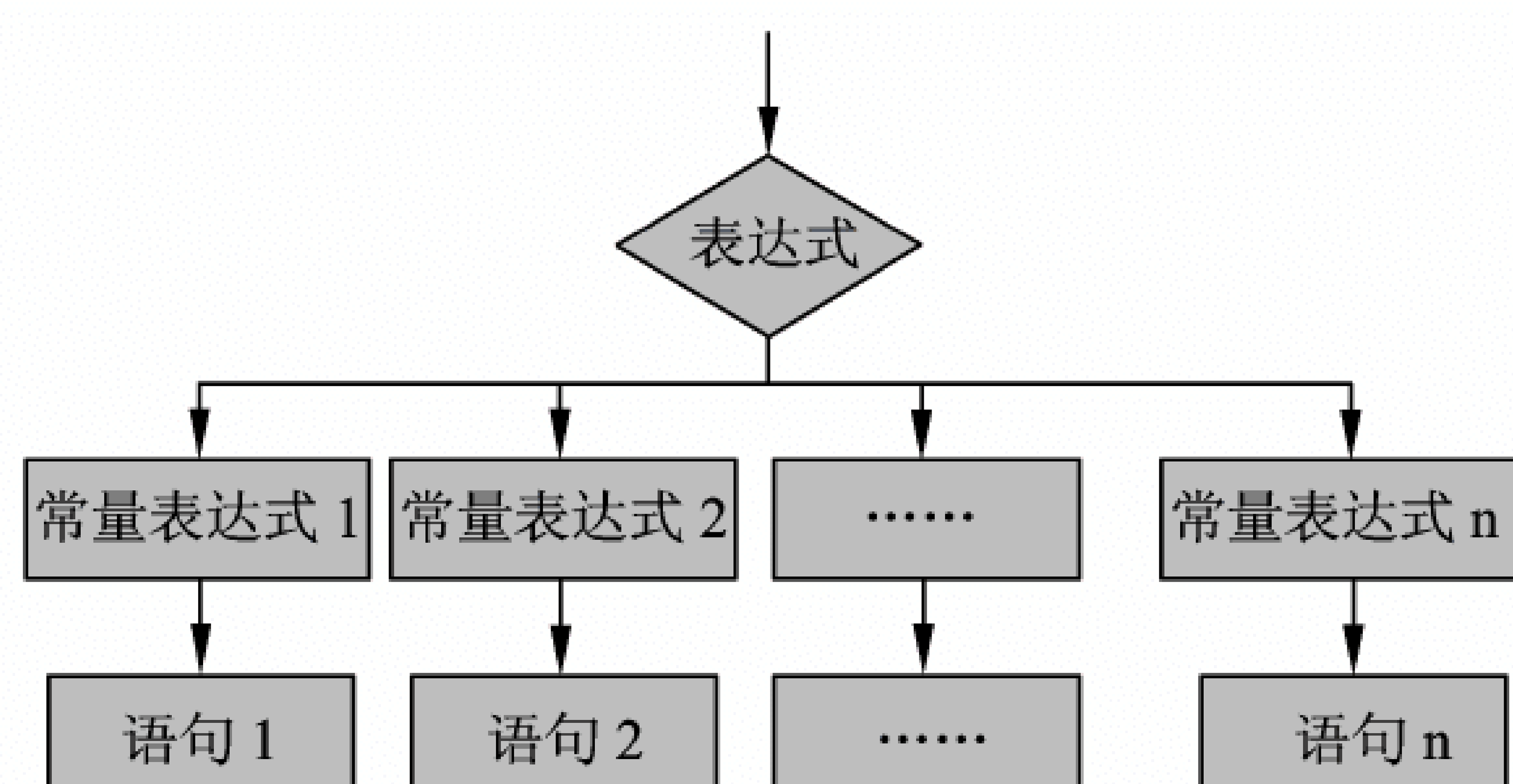


图 3-6　switch 语句流程图

从图 3-6 中可以看出，switch 语句会判断表达式的值，依次判断与各个常量表达式是否相同，与哪个常量表达式相同，就执行相应的语句。其语法格式为：

```
switch ( 表达式 )      //判断表达式的值
{
case 常量表达式 1:     //判断表达式的值是否与常量表达式 1 相同，如果是，则执行语句 1
        语句 1
        break;
case 常量表达式 2:     //判断表达式的值是否与常量表达式 2 相同，如果是，则执行语句 2
        语句 2
        break;
case 常量表达式 n:     //判断表达式的值是否与常量表达式 n 相同，如果是，则执行语句 n
        语句 n
        break;
default:
        语句 n+1
}
```

其中控制代码执行的表达式的值必须使用括号括起来，并且它必须是一个可整型化的类型或者是一个可以明确转换成整型化的类。也就是说，表达式的值必须可以“比较”。switch 语句体根据控制表达式的值、case 标签的值和是否存在 default 标签，确定是无条件地跳转到 switch 语句体还是略过 switch 语句体。其中，case 标签和 default 标签都是可以忽略的。case 标签可以定义多条，每条标签后的取值不能有重复，而 default 是默认情况下要执行的代码，所以最多定义一次。表 3-10 列出了 switch 语句在各种情况下的执行方式。

表 3-10　switch语句的执行方式

条　　件	执 行 方 式
如果转换的 case 标签值与控制表达式的值匹配	程序跳转到 case 标签后的代码段
没有 case 标签值与控制表达式的值匹配，但是 default 标签存在	程序跳转到 default 标签后的代码段
没有 case 标签值与控制表达式的值匹配，同时 default 标签也不存在	程序跳转到 switch 语句后的代码段

在 switch 语句的可能执行的地方，也就是所有可能执行的路径的地方，可以使用带初

始化的定义。在这些地方声明的定义只具有本地作用域，即只能在定义的范围内使用。代码如下：

```
switch( tolower( *argv[1] ) )
{
    //此处程序不可能执行到，声明是无效的
    char szInput[] = "Please Enter Command: ";

    case 'x' :
    {
        //szInput[]的声明有效，是本地范围的变量
        char szInput[] = "Please Enter Command: ";
        cout << szInput << "x\n";  //输出 szInput 数组的值
    }
        break;
    case 'y' :
        cout << szInput << "y\n";  //szInput 未定义
        break;
    default:
       cout << szInput << "既不是 x 也不是 y\n";   //szInput 未定义
       break;
}
```

在上面的代码中，case 'x' 标签下定义的 szInput 变量在本 case 标签下是有效的；第二条 case 标签和 default 标签下的代码段中，调用 szInput 是错误的，因为已经超过了 szInput 的作用范围。

switch 语句与 if 语句一样，也是支持嵌套的，并且 case 标签和 default 标签也是与最近的 switch 语句匹配的。下面的代码是 Windows 消息循环的处理代码段。switch 语句首先判断 msg 的类型，只有当它为 WM_COMMAND 时，才会根据 wParam 参数处理命令，而其中又使用 switch 语句判断命令类型，分别处理 IDM_F_NEW 和 IDM_F_OPEN 命令。代码如下：

```
switch ( msg )                //判断消息类型，进行分类处理
{
case WM_COMMAND:              //Windows 命令，处理多个命令
    switch ( wParam )
    {
    case IDM_F_NEW:           //“新建”菜单命令
        break;
    case IDM_F_OPEN:          //“打开”菜单命令
        break;
    ...                       //此处代码省略
    }
case WM_CREATE:               //创建窗体
    ...                       //此处代码省略
    break;
case WM_PAINT:                //窗体需要重绘
    ...                       //此处代码省略
    break;
default:
    return DefWindowProc( hWnd, Message, wParam, lParam );
                              //处理对话框处理过程
}
```

switch 语句不会在 case 标签和 default 标签中终止语句的运行，要停止 switch 语句的继续运行，则需要在适当的位置加入 break 语句。程序在执行到 break 语句时，会跳转到 switch 语句后的代码。下面的代码说明了 break 语句的使用。

```
BOOL fClosing = false;                      //定义是否关闭对话框的变量
...                                         //此处代码省略
switch ( wParam )                           //判断消息的 wParam 参数
{
case IDM_F_CLOSE:                           //“关闭”菜单命令
   fClosing = true;
case IDM_F_SAVE:                            //“保存”菜单命令
   if( document->IsDirty() )                //判断文档是否修改过
      if( document->Name() == "UNTITLED" )
                                            //如果文档是未命名的新文档，则另存为
         FileSaveAs( document );            //另存文档
      else
         FileSave( document );              //保存文档
   if( fClosing )
       document->Close();   //如果要关闭文档
   break;
}
```

在上面的代码中，当执行 IDM_F_SAVE 命令时，fClosing 为 false，则程序会保存文件，但不会关闭文件；当执行 IDM_F_CLOSE 命令时，程序先将 fClosing 赋值为 true，然后继续执行到 IDM_F_SAVE 标签，保存文档后，关闭文档。这样巧妙的设计，既按照设计完成了功能，又减少了重复代码的工作量。所以在实际编程时，应该巧妙地利用语法原有的特性。

3.6.3　循环语句

循环语句，又称重复语句。它会使语句执行 0 次或多次，直到循环条件不满足。如果循环执行的语句是复合语句，则会按照顺序执行，除非在语句中使用 break 或 continue 等跳转语句，跳转语句在 3.6.4 小节会介绍。

C++中有 3 种循环语句，分别是 while、do 和 for。每种循环语句会重复执行语句，直到终止表达式计算的结果为 false 或者遇到 break 语句强制终止。其中，终止表达式必须是整型化的类型或者是可以明确地转化为整型化的类的表达式，而重复语句的执行语句部分不能是声明语句，但是可以是包含声明语句的复合语句。同时，循环语句也是支持嵌套的，嵌套规则与选择语句相同。循环语句的执行方式如表 3-11 所示。

表 3-11　C++ 循环语句的执行方式

语　句	何时求值	初始化	增　量
while	循环开始	无	无
do	循环结束	无	无
for	循环开始	有	有

1．while循环语句

while 循环语句的语法格式为：

```
while ( 表达式 ) 语句
```

while 语句是在每次执行语句前判断终止条件是否为 true。如果为 false，则退出循环语句；如果为 true，则继续执行循环语句，因此 while 循环有可能执行多次，也有可能一次也不执行。其流程图如图 3-7 所示。

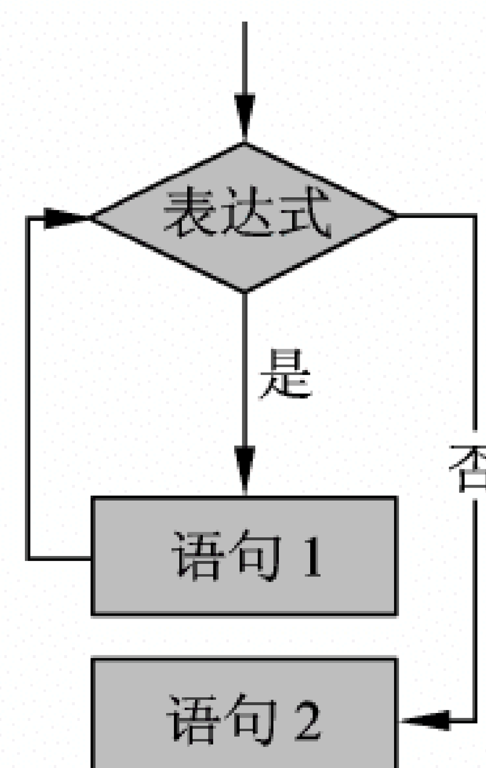


图 3-7　while 语句的执行流程图

从图 3-7 中可以看出，程序首先判断 while 后的表达式的值是否为 true，如果为 true，则执行语句 1，执行完后会继续判断表达式，直到表达式的值为 false，则会执行语句 2。如下代码显示了 while 语句的使用。

```
char *trim( char *szSource )                              //去掉字符串头尾的内容
{
    char *pszEOS;                                         //定义字符指针
    pszEOS = szSource + strlen( szSource ) - 1;           //设置开始指针在字符串尾
    while( pszEOS >= szSource && *pszEOS == ' ' )         //循环处理字符串中的字符
        *pszEOS-- = '\0';                                 //去掉空格字符
    return szSource;                                      //返回处理后的字符串
}
```

上面代码的功能是去掉字符串尾部的空格。代码首先将指针指向要处理的字符串的最后一个字符的地址处，然后开始执行循环，去掉尾部的空格，直到指针到达字符串头，或者最后一个字符不是空格，即退出循环。对于此函数，如果字符串尾部没有空格，则循环会退出。

2．do循环语句

do 循环语句的语法格式如下：

```
do 语句 while （表达式）；
```

do 语句重复的执行语句，直到指定的终止条件表达式计算结果为 0 时，即退出循环。终止条件的判断是在每次循环语句执行完一次循环后执行，因此 do 循环语句至少要执行一次。其流程图如图 3-8 所示。

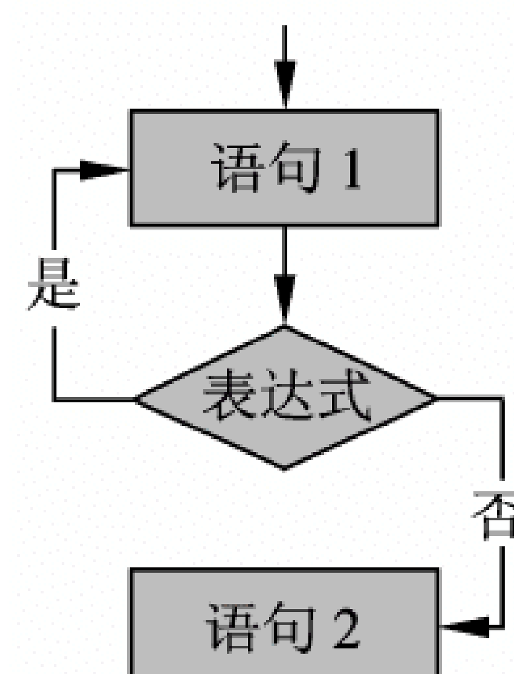


图 3-8　do 语句的执行流程图

从图 3-8 中可以看出，程序首先执行语句 1，然后判断 while 后的表达式的值是否为 true。如果为 true，则继续执行语句 1；否则会执行语句 2。如下代码显示了 do 语句的使用。

```
void WaitKey( char ASCIICode )        //等待用户输入指定字符的函数
{
    char chTemp;                      //定义字符变量，用于存放用户输入的字符
    do
```

```
    {
        chTemp =  getch();            //接收用户输入的字符
    }
    while( chTemp != ASCIICode );     //循环条件为判断接收到的字符是否为指定字符
}
```

上面的函数实现了等待用户按下指定按键的功能。它会提示用户输入字符，直到输入的字符与要求的字符相同，则会退出 do 循环。此功能也可以使用 while 循环语句实现，但是实现起来就比较繁琐，没有 do 循环语句简明。可以看出，在至少需要执行一次的循环语句中使用 do 循环语句比较适合，否则还是使用 while 语句比较好。如下代码使用 while 语句改写了上面的代码。

```
void WaitKey( char ASCIICode )        //等待用户输入指定字符的函数
{
    char chTemp;                      //定义字符变量，用于存放用户输入的字符
    chTemp = _getch();                //接收用户输入的字符
    while( chTemp != ASCIICode )      //循环条件为判断接收到的字符是否为指定字符
    {
        chTemp =  getch();            //接收用户输入的字符
    }
}
```

3．for循环语句

for 循环语句可以分为 3 部分，其语法格式如下：

```
for （表达式 1；表达式 2；表达式 3） 语句
```

其中，表达式 1 部分用于初始化循环参数，是在执行 for 循环语句前，首先执行的语句。表达式 2 部分是整型化的表达式或可以转换成整型化的类。在每次执行循环语句前，判断表达式 2 的值是否为 true，决定是否继续执行。表达式 3 部分用于处理循环计数，每次执行循环语句完成后执行此部分，执行完后进入下一次循环。如图 3-9 显示了 for 循环的执行流程。

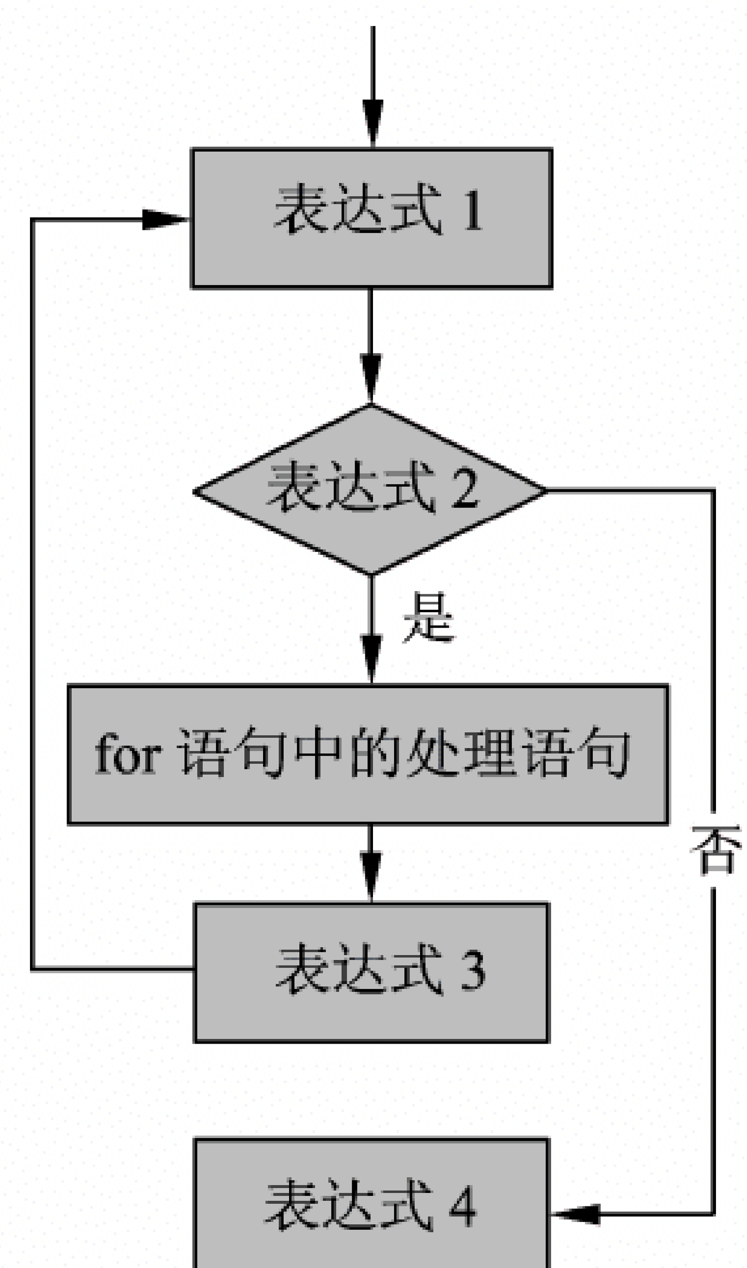


图 3-9　for 语句的执行流程图

for 初始化语句通常用于声明和初始化循环索引变量，判断语句用于测试循环终止条件，循环处理计数语句用于增加循环计数。在 for 循环中，这 3 条语句任何一条都是可选的。其中，for 初始化语句可以是声明语句或表达式语句，也可以是空语句，并且可以包含多条，其间以逗号分开。注意任何在 for 初始化语句中声明的对象，都是在 for 本地作用域内有效。因为 for 语句和 while 语句都是循环语句，因此，二者之间是可以互相转换的。如下面的 for 语句与 while 语句就具有相同的作用。

```
for(表达式 1；表达式 2；表达式 3)
{
    //语句
}
```

与

```
表达式 1;
while(表达式 2)
{
    //语句
    表达式 3;
}
```

而下面的 for 循环语句与 while 循环语句都能够完成无限循环的功能。

```
for( ; ; )
{
    //要执行的语句
}
```

与

```
while( 1 )
{
    //要执行的语句
}
```

虽然通常情况下，for 语句的 3 个语句分别用于初始化、测试终止条件和递增计数，但是也可以不这样用，如下代码，作用是打印 1～100 之间的整数，其 for 语句的执行语句为空语句，在 for 语句的表达式语句中完成打印。

```
void main()
{
    for( int i = 0; i < 100; cout << ++i << endl )      //执行 for 循环
        ;
}
```

3.6.4 跳转语句

跳转语句控制代码直接跳转到指定的代码处。C++中有 4 条跳转语句，分别是 break 语句、continue 语句、return 语句和 goto 语句。

1. break语句

break 语句用于退出循环语句或 switch 选择语句，它会控制程序直接跳转到紧跟在循环语句或 switch 语句后的代码语句上。break 语句仅会终止离它最近的循环语句或 switch 语句。在循环语句中，break 语句用于提前退出循环语句。而在 switch 语句中，break 语句用于终止代码段，通常用在 case 标签前。如下代码显示了如何在 for 循环语句中使用 break 语句终止循环语句的执行。

```
for( ; ; )                  //没有终止条件
{
     if( List->AtEnd() )
         break;             //判断是否到达链表尾，如果到达链表结尾，则退出 for 循环
    List->Next();           //取下一个链表元素
}
```

2．continue语句

continue 语句强制代码跳转到离它最近的循环继续语句中，即强制开始一次新的循环。因此，continue 语句只能依赖于循环语句。在 for 循环语句中，执行 continue 语句，也就是执行一次“表达式 2”，然后执行“表达式 3”。

```
//获取输入的合法字符在字符串中的位置
int GetLegalChar( char *szLegalString )
{
    char *pch;                      //定义指向字符的指针
    do
    {
        char ch = _getch();         //获取用户输入的字符
         //判断输入的字符是否在传入的字符串中存在
        if( (pch = strchr( szLegalString, ch )) == NULL )
            continue;               //如果没有则使用 continue 语句，重新执行循环
        return (pch - szLegalString);//返回合法字符在字符串中的位置
    } while( 1 );                   //执行循环条件
    return 0;                       //返回
}
```

3．return语句

return 语句使函数直接返回到调用它的函数中去，对于主函数来说，return 语句会使程序退出。可以使用表达式，将 return 值传回给调用函数。但是对于函数类型为 void，构造函数和析构函数来说，不能通过 return 语句指定返回表达式。其他类型的函数，必须使用 return 语句指定返回值。在一个函数中可以多次使用 return 语句。如果指定了表达式，则会转换成函数声明的返回值类型。

4．goto语句

goto 语句可以无条件地将程序跳转到标签标识的代码段中。要使用 goto 语句，必须在要跳转到的代码前使用标签标注出来。声明方法为在程序源代码前使用标识符，在标识符后加一个冒号。这样，在代码中就可以使用 goto 加标识符，从而直接跳转到指定的代码部分。如下代码所示。

```
for( p = 0; p < NUM PATHS; ++p )        //使用 for 循环处理文件
{
    NumFiles = FillArray( pFileArray, pszFNames )  //使用文件填充数组
    for( i = 0; i < NumFiles; ++i )     //使用 for 循环打开文件
    {
        if( (pFileArray[i] = fopen( pszFNames[i], "r" )) == NULL )
                                        //打开文件
            goto FileOpenError;         //打开文件失败跳转到 FileOpenError 处
                                        //处理打开的文件
    }
}
FileOpenError:
    cerr << "打开文件错误，中断处理。\n" );
```

在上面的代码中，当打开文件失败时，程序会直接跳转到最后的 FileOpenError 标签的

代码处，打印出错误信息。标签不能单独出现，其后必须要加上语句，如果没有需要处理的语句，则可以加上一条空语句。标签必须在当前函数内，作用范围也仅限于当前函数。它在同一函数内不能重复声明，但是在不同的函数内可以使用相同的标签名称。

3.7　函　　数

函数是 C++语言的核心组成部分，是完成一定功能的语句和变量的组合。根据函数作用的范围和功能的不同，函数分为很多种，如全局函数、类的构造函数、类的析构函数和内联函数等。本节将主要介绍函数的使用方法及需要注意的问题。

3.7.1　函数的定义和调用

函数是完成特定功能的代码段，需要使用函数名称标识代码段，可以不使用参数，也可以使用多个给定类型的参数；返回值可以是指定类型也可以不返回任何类型，此时返回值为 void。函数定义的语法为：

```
可选的说明符 函数名称 (参数列表) 其他函数限定符
{
    //函数体
}
```

其中，可以在可选的说明符中指定函数的返回类型。函数名称指定了函数的名字，不能与相同作用域中的其他标识符重名。参数列表是可选部分，函数可以不指定任何参数，参数列表中各个参数之间使用逗号分隔开，每个参数需要指定参数的类型。其他函数限定符可以是 const 关键字或 volatile 关键字，分别表示常量函数和变化函数。需要注意的是，函数在定义之前，需要先声明，声明的语法就是上面中的第一行，并在行尾加上分号即可。以下代码是一个函数定义的示例。

```
int Add(int a, int b )          //加法函数定义
{
    return a+b;                 //返回两个参数的和
}
```

上面的 Add()函数定义了实现将两数相加的功能，其中 Add 为函数名称。第一个 int 表示函数的返回类型为整型 int。小括号里面表示此函数具有两个参数，都是 int，分别为参数 a 和参数 b。在函数体中，编写代码返回整型参数 a 和整型参数 b 的和。

函数调用是调用函数执行的表达式，其语法为：

```
函数标识符 ([参数列表])
```

其中，函数标识符是指要调用的函数名称或是指向函数的指针值。参数列表是函数调用中的可选部分，是传入函数的参数值的表达式集合。当忽略参数列表时，表示函数的参数部分为空。参数列表可以包含一个或多个参数，各个参数之间用逗号分隔开，传入的参数类型必须与函数定义中相应位置的参数类型相同。

函数调用表达式具有返回值，类型与函数的返回值类型相同。函数不能返回数组类型

的对象。如果函数的返回值类型为 void，也就是说，函数声明为不返回值的形式，则函数调用表达式也具有 void 类型。

```
int c = Add(2 , 3);//函数调用示例
```

上面代码表示创建整型变量 c，存储整型值 2 和整型值 3 的和。

3.7.2　带默认形参值的函数

有时候，函数会具有一些参数，这些参数通常情况下取值是确定的，只需要在特殊情况下修改传入的参数。这时，就需要使用带默认形参值的函数。使用参数默认值，必须保证指定的默认参数值是有效取值。其语法格式为：

```
可选的说明符 函数名称 (参数列表，参数类型 形参名称=形参值，...) 其他函数限定符
{
    //函数体
}
```

带默认值参数的函数的定义与普通函数的定义方式是相同的。区别在于，需要在函数定义中具有默认值的参数后加上等号（=）及其默认值。以下代码为带默认值参数的函数的定义方法。

```
int AddYear(int old, int increment = 1)            //为时间值增加一年
{
    return old + increment;
}
```

上面的代码定义了 AddYear()函数，其返回值为 int，表示处理后的年份的取值。此函数有两个参数，第一个参数是要进行年处理的原来的年份值，第二个参数为要在基础年的基础上增加的年数，由于默认情况下每次增加一年，所以，此参数的默认值为 1。

调用带默认参数值的函数的方法与调用普通函数的方法相同。区别在于，它可以使用两种方式调用函数，一种方式是按照参数列表中的参数依次输入各个参数；另一种方式是只传入普通参数，不为带有默认值的参数赋值，此种情况下，对应参数会采用参数的默认值进行处理。代码如下：

```
//带默认参数的函数调用示例，不使用默认参数值，结果 c=2010
int c = AddYear(2008 , 2);
//带默认参数的函数调用示例，使用默认参数值，结果 d=2009
int d = AddYear(2008);
```

上面的代码显示了如何使用带默认值参数的函数的用法，第一种调用方式不使用默认参数值，函数返回结果是两个参数都起作用的结果，结果为 2010；第二种调用方式使用参数默认值，函数返回结果为第一个参数值与第二个参数的默认值的计算结果，结果为 2009。使用带默认形参值的函数需要注意以下几个问题。

- 带有默认值的参数必须放在函数参数列表中的结尾处，并且在调用函数时，传入的参数值，也是从左到右赋值的。以下代码就是不可用的：

```
int AddYear(int old=2008, int increment){}
```

- 带默认参数值的函数，其中的参数默认值不仅可以是常量，还可以指定参数表达式，如可以使用函数返回的值。下例是创建滚动条的函数声明，使用 Win32 API 函数 GetSystemMetrics()函数来获取高度值。

```
BOOL CreateHScrollBar(HWND hWnd,  short nHeight=GetSystemMetrics(SM_
CYHSCROLL ) );
```

3.7.3　函数的递归调用

递归函数是指在函数内部调用函数自身的函数。理解递归函数最好的例子就是使用阶乘的概念，阶乘就是计算从 1 到给定数之间的所有整数的乘积。递归是一种非常重要的技术，但是递归的使用要尤其注意不能出现无限递归的情况。当函数无法获取确定的结果，或者无法计算到结束点时，就会出现无限递归，从而导致无限循环。当出现这种情况时，程序的逻辑结构一定是有问题的。因此，在设计递归函数时要尤其小心。下面是实现阶乘计算函数的代码。

```
01  #include <iostream>
02  using namespace std;
03
04  long factorial(int number)      //递归函数的调用，功能是计算指定整数的阶乘
05  {
06      if (number < 0)
07          return -1;       //如果传入的参数小于 0，则无法计算阶乘
08      if ((number == 0) || (number == 1))
09          return 1;        //如果传入的参数等于 0 或 1，则阶乘为 1
10      else                 //否则，递归调用，直到计算完成
11          return (number * factorial(number - 1));    //递归调用阶乘函数
12  }
13  void printRecuFunction()                           //使用递归函数的示例
14  {
15      int a;                                          //定义整型变量 a
16      cout << "递归示例，输入要计算阶乘的数：";           //输出提示信息
17      cin >> a;                                       //接收用户输入
18      cout << a << "的阶乘 " << factorial(a) << "\n";//输出计算的阶乘结果
19
20  }
21  int main()                                          //程序入口函数
22  {
23      printRecuFunction();                 //执行 printRecuFunction()函数
24      getchar();                           //阻止控制台程序自动退出
25      getchar();
26      return 0;                            //返回
27  }
```

在上面代码中，计算阶乘值的 factorial()函数就是一个递归函数，因为，在函数中调用了函数本身 factorial，用于与计算比当前函数值小 1 的数的阶乘，直到计算到 1 的阶乘。此处需注意，在 factorial()函数中，首先判断了传入的参数是否为负数，如果是负数，则返回错误。这是因为如果不判断参数是否为负数，则计算–5 的阶乘时，会首先计算–6 的阶乘，而计算–6 的阶乘时，首先计算–7 的阶乘……这样下去，就会进入无限递归的情况，程序永

远执行不完。运行结果如图 3-10 所示。

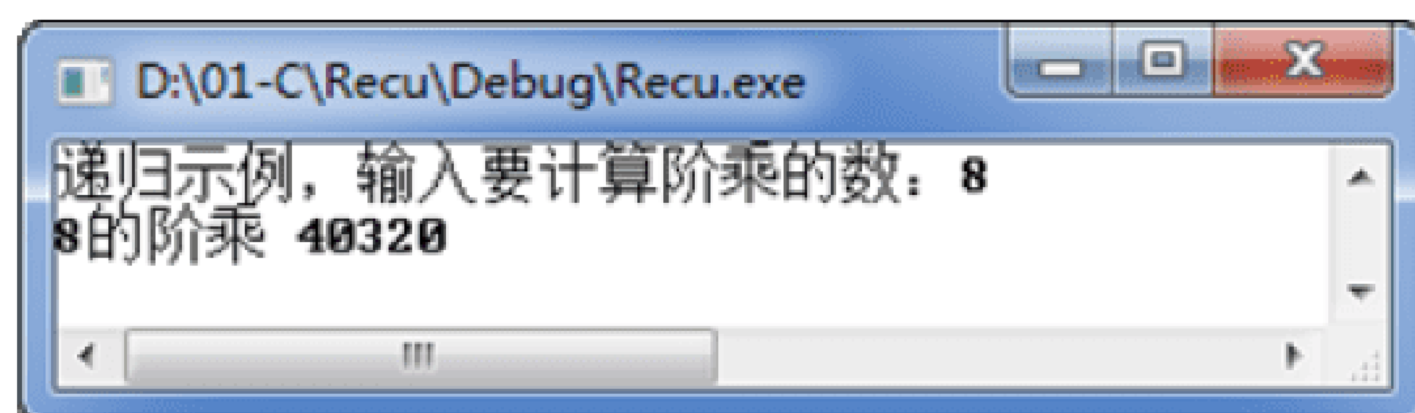


图 3-10　递归函数运行结果

上面的结果显示了如何调用计算阶乘的函数，当输入要计算的值 8 时，计算结果为 40320。

3.7.4　内联函数

inline 关键字用于指定内联函数，告诉编译器使用函数体的代码代替函数调用部分。此种替换方式也就是“内联展开”，也称为内联。内联展开通过增加代码大小来减轻函数调用的系统开销。inline 关键字告诉编译器优先进行内联展开。然而，编译器可以创建函数的单独实例，并创建标准的调用链接代替插入内联代码，从而适当减少内联的代码量。以下代码是内联函数的例子。

```
inline int max( int a , int b )          //取两个值中较大的一个
{
    if( a > b )
        return a;                        //如果 a 大于 b，则返回 a 的值
    else
        return b;                        //否则，返回 b 的值
}
```

上面的代码是取两个数中较大的一个的函数，此函数通过 inline 关键字定义为内联函数。除了可以指定一般函数为内联函数外，类的成员函数也可以声明为内联函数。有以下两种方法可以声明类的成员函数为内联函数。

- 显式指定内联函数：在类的成员函数的定义前面加上 inline 关键字。
- 隐式指定内联函数：在类的成员函数的声明中，直接加上函数体的内容。

下面代码显示了指定类的成员函数为内联函数的方法。

```
class MyClass                                //自定义 MyClass 类
{
public:
    int min()                                //隐式内联
    {
        if( a < b )
            return a;
        else
            return b;
    }
    int max();                               //取较大值函数
private:
    int a;                                   //整型变量值 a
    int b;                                   //整型变量值 b
```

```
};
inline int MyClass::max()                         //显式内联，取较大值函数
{
    if( a > b )
        return a;                                 //如果 a 大于 b，则返回 a 值
    else
        return b;                                 //否则返回 b 值
}
```

在上面的代码中，MyClass 类使用显式方式定义 max()函数为内联函数，使用隐式方式定义 min()函数为内联函数。

3.7.5　函数的重载

C++允许在同一作用域内指定相同函数名的多种函数形式，此技术称为函数重载。重载函数允许程序员根据参数类型和个数提供函数的不同语义。对程序员来说，函数重载就是定义相同函数名的多个函数原型，但是重载函数的返回值类型必须相同。

- 可以使用参数个数进行函数重载，即函数名称和返回值相同，但是参数的个数不同。
- 可以使用参数类型进行函数重载，即函数名称、返回值和参数个数都相同，但是参数的类型不同。
- 可以使用是否设置参数的默认值进行函数重载，即 3.7.2 小节中使用的 AddYear()函数的例子。
- 可以使用 const 和 volatile 关键字重载函数。

以下代码中定义的重载函数可以实现在多种数据类型中获取较大值的功能。

```
void max(int a, int b, int& c);
void max(double a, double b, double& c);
void max(char a, char b, char& c);
```

3.8　指针和引用

为了提高存储效率，C++中提供了指针和引用两种类型。这两种类型也是 C++数据类型中的重点和难点。在学习本节内容时，一定要透彻理解指针和引用的概念，在运用时要能够融会贯通、举一反三。

3.8.1　指针和指针变量

不管在 C 还是 C++中，指针类型都是一个难点和重点。C++中的指针指向内存中一个变量或对象的存储空间。因此，指针的定义分为两种情况，一种是指向给定类型的变量的指针，一种是指向类成员的指针。

指针变量指定变量指向的对象的类型。对象可以是全局的、本地的或者动态分配的。

指针类型指定了指针指向的对象类型，可以是基本类型、结构或联合体。指针变量也可以指向函数、数据和其他指针。使用指向给定类型的函数的指针，可以在程序运行时确定指定对象选择的函数。下面的代码是有关指针声明的几个例子。

```
char *szNameStr;                //指向 char 类型的指针
int *iCount;                    //指向 int 类型的指针
int const *x;                   //声明指针变量 x，指向一个常数值
int *const y = &a;              //声明常指针变量 y，指向一个 int 值
int *const volatile z = &b;     //声明固定值的常指针变量
```

在上面语句中，第一条语句声明了一个指向 char 类型的指针。第二条语句声明了一个指向 int 类型的指针。第三条语句声明了指针变量 x，可以修改其指向不同的 int 类型值，但是指向的 int 值是不能修改的。第四条语句声明的指针变量 y 是常指针，可以修改其指向的 int 类型的值，但是不能修改其指向其他的 int 值，即只能指向对象 a 的地址。第五条语句声明了指针变量 z，程序既不能修改其指向的值，也不能修改指针本身的取值。

3.8.2　&和*运算符

地址操作符符号为&，是一目运算符，用于获取操作数的地址。地址操作符的操作数可以是函数定义或是表示对象的变量，但不能是位字段，也不能是使用 register 声明的存储类关键字。地址操作符应用于基本类型变量、结构变量、类变量或共用体变量。代码如下：

```
void printAddressOf()               //地址操作符的示例
{
    int *pPtr;                      //定义整型指针变量 pPtr
    int nArray[5];                  //定义整型数组
    pPtr = &nArray[2];              //赋值指针变量指向整型数组的第二个元素
    cout << pPtr << "\n";           //输出指针变量的值
    cout << &nArray << "\n";        //输出数组的地址值
}
```

上面的代码使用地址操作符获取 nArray 数组的第三个元素的地址，并将其存储在 pPtr 指针变量中，然后将其输出，并输出 nArray 数组的起始地址。运算结果为：

```
0x0012FEE0
0x0012FED8
```

从上面的运行结果可以看出，因为数组元素是整型，所以每个元素占用 4 个字节。数组起始地址为 0x0012FED8，也就是第一元素的存储地址，则第二个元素的存储地址为 0x0012FEDC，因此，第三个元素的存储地址为 0x0012FEE0，如图 3-11 所示。

注意：此处使用的地址值是根据内存情况由系统分配的，当每次测试时，结果值可能都不相同，这里仅是举例说明。

指针操作符符号为*，也称为间接访问操作符，是一目运算符，通过指针间接地获取操作数的取值。它的操作数必须是指针值，运算结果为操作数指向的地址中存放的值，代码如下：

```
void printIndirection()                //间接操作符的示例
{
    int nTest;                         //定义整型变量 nTest
    int *pTest;                        //定义指针
    pTest = &nTest;                    //为指针变量分配值为 nTest 的地址
    *pTest = 15;                       //为 pTest 指向的存储区赋值
    cout << nTest << "\n";             //输出 nTest 的值
}
```

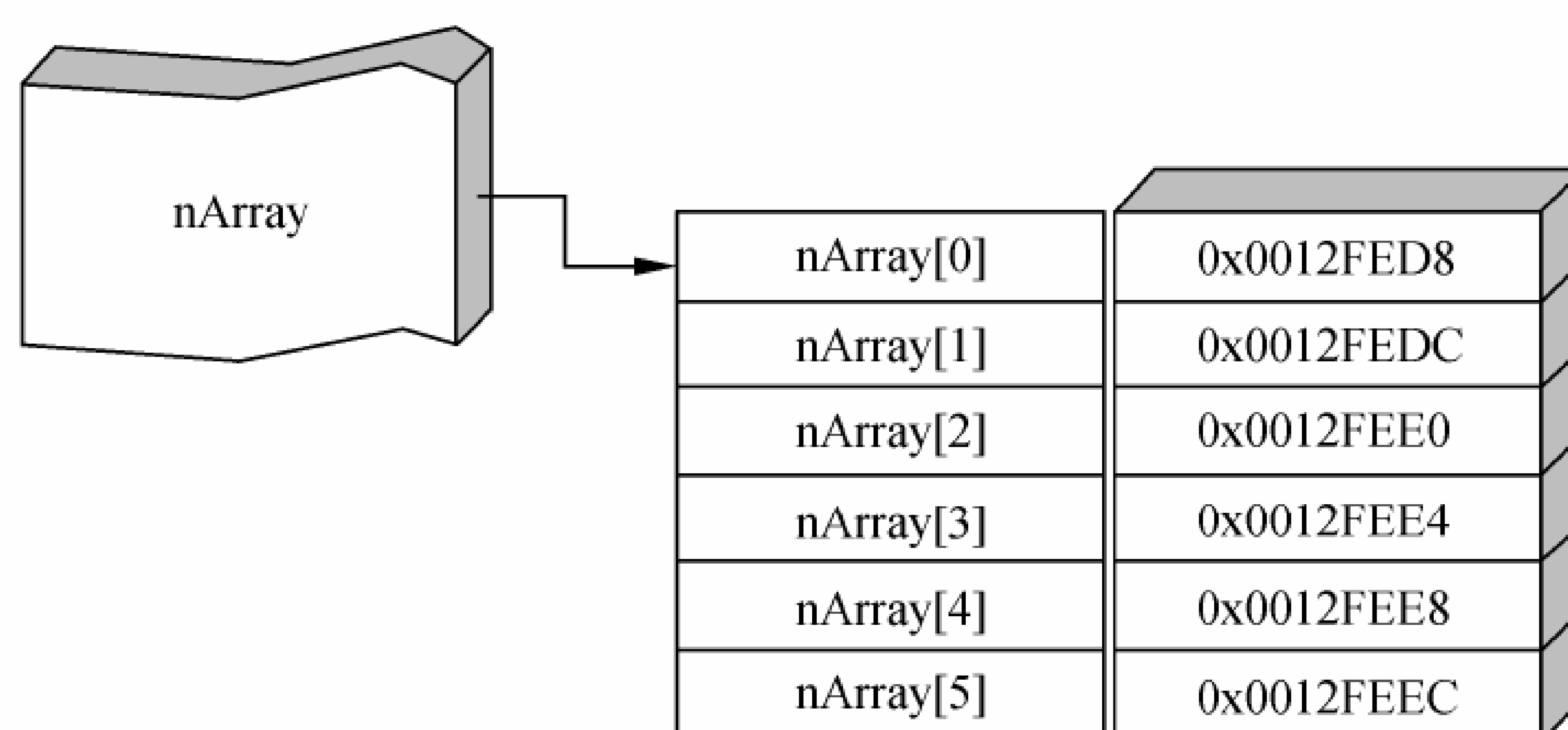


图 3-11　数组地址示意图

上面代码定义了两个变量，一个是整型类型的变量 nTest，一个是指向整型变量的指针类型 pTest。首先将整型变量的地址赋值给整型指针变量，此时，pTest 中存储的是 nTest 的地址。然后使用*间接操作符将 pTest 指向的地址中的值赋值为 15，最后输出 nTest 的值。运算结果为：

```
15
```

从上面的运行结果可以看出，*运算符可以看作获取操作数中存储的地址值中的实际值的运算符。虽然&和*符号的含义简单，但是只有深刻理解其本质，才能处理各种复杂的组合情况。

3.8.3　指针和数组

C++中允许指针指向数组。以下代码用来声明指向数组的指针。

```
int *student[10];              //声明指针数组，数组中的每个元素为指向整型的指针
int (*student )[10];           //声明一个指针，指向一个具有 10 个元素的数组
```

上面这两条声明的变量是不同的，第一条是声明了一个指针数组，它是一个具有 10 个元素的数组，其中每个元素都是一个指向 int 类型的指针。第二条声明了一个指针，它指向一个具有 10 个元素的数组。所以，要注意当指针遇到数组时的处理。

3.8.4　指针和结构体

C++中允许指针指向结构体，并且在结构体、共用体或枚举类型定义之前，可以使用标记声明指向结构体、共用体或枚举类型的指针。对于指针变量编译器不需要预先确定指

向的结构体或占用存储空间大小的共用体，因此使用指针定义结构体效率相对较高。代码如下：

```
struct collection *next, *previous;     //使用 collection 标记定义指针
struct collection                       //collection 结构的定义
{
   char *student;                       //存放学生名称的字符串
   int age;                             //存放学生年龄
   struct collection *next;             //指向下一个学生结构
}students;
```

上面代码声明了名为 next 和 previous 的两个指针变量，均指向结构体 collection。这条声明可以在 collection 结构定义之前声明，当定义结构后，则在声明中结构也就可见了。变量 students 是 collection 结构类型的变量。collection 结构具有 3 个成员，第一个数据成员是指向 char 类型的 student 指针，用于存储学生代码；第二个数据成员是指向 int 类型的 age 指针，用于存储学生年龄；第三个数据成员是指向另外一个 collection 结构的指针，用于链接下一条学生记录。

3.8.5　函数的指针传递

在函数中可以像使用其他类型一样，将指针作为函数参数传递。示例代码如下：

```
struct student                          //学生结构
{
   char name[50];                       //存放学生名称的字符串
   int age;                             //存放学生年龄
};
void printStudent(student* stu)         //输出学生信息
{
    cout << "姓名=" << stu->name << ";年龄=" << stu->age <<".";
                                        //输出学生姓名和年龄
}
void TestPassPoint()                    //测试指针传递
{
    student* stu = new student();       //创建指向 student 结构的变量
    sprintf(stu->name, "%s", "张三");   //为 student 结构的 name 分量赋值为张三
    stu->age = 10;                      //为 student 结构的 age 分量赋值为 10
    printStudent(stu);                  //将 stu 指针传入 printStudent 参数，
                                        //输出学生信息
}
```

上面代码定义了 student 结构体和打印学生信息的 printStudent()函数。在 TestPassPoint()测试函数中，定义了 student 类型的变量并赋值，将其传入 printStudent()函数中，从而输出定义的学生信息。

3.8.6　引用及函数的引用传递

引用类型（&）是一个 16 位或 32 位的存放对象指针数，但是它的行为更像对象。引用声明的格式如下：

```
声明标识符 & [限定符列表] 引用名称 ;
```

其中，声明标识符列出了要定义的数据类型信息，限定符列表是一个可选的包含引用定义的限定符选项，引用名称是定义的引用的名称。在 C++中，任何地址都可以转换为指针类型的对象，也可以转换成对应的引用类型。如任何地址都可以转换为 char*的对象，也可以转换成 char&类型。以下代码显示了引用类型的使用。

```
void AddOne(int& a)                          //引用类型示例
{
    a++;                                     //将传入的变量的值自增 1
}
void printRefrence()                         //打印通过引用传递的值
{
    int x= 99;                               //定义整型变量 x，赋值为 99
    cout << "x=" << x << "\n";               //输出 x 的值
    AddOne(x);                               //通过调用带有引用参数的函数改变变量的值
    cout << "AddOne(x) x=" << x << "\n";    //输出改变值后的 x 的值
}
```

在 AddOne()函数的定义中，int 是声明标识符，&是引用操作符，x 是定义的引用名称。在 printRefrence()函数中，定义了 int 类型的 x，并将其传入 AddOne()函数中，在 AddOne()函数中，将传入的变量 a 的值自增 1。在 AddOne()函数中修改的 a 的值，就是修改了 printRefrence()函数的 x 值。因此，&操作符与地址操作符的功能是相同的，它传入的是变量的引用，而不是一般函数的形参。函数的运行结果如下：

```
x=99
AddOne(x) x=100
```

3.9　预　处　理

C++程序进行编译前，会首先检索程序代码，将其中的预处理指令进行转换，然后将转换后的代码提交给 C++编译器，编译器再进行程序编译，这个预处理指令转换的过程称为预处理。在 C++中预处理主要包括 3 种，分别是宏定义、文件包含和条件编译。本节将分别介绍这 3 种预处理方式。

3.9.1　宏定义

C++中，预处理通过预处理指令完成，所有的预处理指令都是以#开头。宏定义的作用是为常用的对象分配一个有意义的名称，指令是#define，语法格式如下：

```
#define 宏名称 宏内容
```

宏定义指令定义的内容会在预处理时，将所有使用宏名称的地方都使用宏内容代替。但是当宏名称出现在注释中或包含在其他标识符中时，预处理器不会替换宏内容。在宏内容中可以包括一系列标记，如关键字、常数、完整语句和空格等。宏名称中可以使用参数，但是在使用宏时，传入的参数必须与宏定义中指定的参数个数一致。在定义具有参数的宏

时，最好使用小括号分隔宏名称和宏参数，使得代码清晰易懂。以下代码为宏定义的示例。

```
#define LENGTH  100          //定义名称为 LENGTH 的宏，值为 100
#define ADD(a,b)  (a + b)    //定义名称为 ADD 的宏，作用是将 a 和 b 相加
#define min(a,b) \
    (if (a>b) ? b : a )      //定义名称为 min 的宏，作用是获取两个值中较小的一个
```

上面的代码定义了 3 个宏，分别是 LENGTH、ADD 和 min。其中，LENGTH 宏代表一个常量，ADD 宏代表一个加法表达式，min 宏代表一个关系运算表达式。在定义 min 宏时，使用反斜线作为续行符。

在 C++中，可以使用#if defined 和#ifdef 指令判断是否定义过指定的宏。使用#undef 指令可以关闭已经使用#define 指令定义的宏。其语法格式为：

```
#undef 宏名称
```

其中，宏名称表示要取消的宏的名称。以下代码会取消对 LENGTH 宏的定义，在后面代码中如果使用 LENGTH 宏将会产生错误。

```
#undef LENGTH
```

3.9.2　文件包含

在 C++程序中，一个系统可以由多个程序模块组成，每个程序模块由多个文件组成，这就会出现在一个文件中引用其他文件中的内容的情况，此时需要使用#include 文件包含预处理。#include 指令告诉预处理器在使用此指令的文件中包含指定文件中的代码。

文件包含支持嵌套，即被包含的文件中也可以使用#include 指令包含其他文件。可以将用到的常数和宏定义放置到文件中，然后使用#include 指令将这些定义添加到任何需要使用这些常数和宏的源文件中。#include 指令对于组织外部变量和复杂的数据类型来说非常有用。使用#include 指令使得在多处使用的定义可以只在一处定义。#include 指令的语法格式如下：

```
#include "文件名"
#include <文件名>
```

其中，文件名表示要加入内容所在的文件名，可以指定文件所在路径。文件名的语法格式根据操作系统的不同会有差别。#include 的两种语法格式的功能是相同的，区别仅在于当没有指定完整的文件路径时，预处理器查找头文件的路径有所不同。

- ❑ 当使用第一种语法指定文件包含时，会首先从包括#include 语句的文件所在的路径下查找，然后从其他包含此文件的文件所在的路径查找，再从/I 编译选项指定的路径中查找，最后会在 INCLUDE 环境变量中指定的路径中查找。
- ❑ 当使用第二种语法指定文件包含时，会先从/I 编译选项指定的路径中查找，然后在 INCLUDE 环境变量中指定的路径中查找。

以下代码是#include 指令的使用代码。

```
#include <stdio.h>
#include "Sample.h"
```

上面的代码包含 stdio.h 文件和 Sample.h 文件。其中，stdio.h 文件会在/I 编译选项指定的路径和 INCLUDE 环境变量中指定的路径中查找，而 Sample.h 会首先从文件本地路径中查找。

3.9.3　条件编译

C++支持条件预编译指令控制源文件编译的部分。条件编译指令有#if、#elif、#else 和#endif 指令。如果编译指令#if 后的表达式为非 0 值，则其后的代码会进行编译，直到#endif 指令结束处。在源文件中，每个#if 指令必须与#endif 指令配对。在#if 和#endif 之间可以使用多个#elif 指令判断多种条件，但是其中最多只能有一条#else 指令。另外，如果使用#else 指令，则必须是#endif 指令前的最后一条指令。条件编译指令是支持嵌套的，每个嵌套的#else、#elif 和#endif 指令与距离其最近的#if 指令配对。

通常情况下，条件编译会与#defined 配对使用，判断指定的宏名称是否已经定义，确定是否编译指定代码。代码如下：

```
#if defined(DB_SQLSERVER)          //如果定义了 DB_SQLSERVER
    connectSQLServer();            //连接 SQLServer 数据库
#elif defined(DB_ACCESS)           //如果定义了 DB_ACCESS
    connectACCESS();               //连接 Access 数据库
#else                              //如果没有定义数据库类型
    printerror();                  //输出错误信息
#endif
```

在上面代码中，首先使用#if defined(DB_SQLSERVER)语句判断是否定义了 DB_SQLSERVER 宏。如果定义了该宏，会使用 connectSQLServer()函数连接 SQLServer 数据库；如果没有定义 DB_SQLSERVER 宏，则继续判断是否定义了 DB_ACCESS 宏。如果定义了 DB_ACCESS 宏，则会使用 connectACCESS()函数连接 Access 数据库，否则，会报告错误。

3.10　文件操作

文件是操作系统组织数据和操作的基本元素。计算机从最初的科学计算向前迈进的第一步就是文件系统。文件将数据和操作分单元存储，可以写入数据、读取数据等。因此，对文件的操作也主要分为打开文件、读文件、写文件和关闭文件。本节将介绍有关文件的操作。

3.10.1　打开文件

要对文件进行读写，首先需要打开文件，使用 fopen()函数和_wfopen()函数可以打开文件。其原型为：

```
FILE *fopen( const char *filename,       //表示要打开的文件名称
             const char *mode );         //打开文件时的访问权限类型
```

```
FILE *_wfopen( const wchar_t *filename, const wchar_t *mode );
```

其中，mode 参数的有效取值如下。

- r：读取文件打开。如果文件不存在或查找不到，则 fopen()函数调用失败。
- w：打开空文件写入。如果指定的文件存在，则内容会被删除。
- a：打开文件，从文件尾开始添加内容，在向文件中写入新数据前不删除 EOF 标记。如果文件不存在，则创建文件。
- r+：打开文件，既可以从文件中读取，也可以向文件中写入，但是文件必须存在。
- w+：打开空文件，既可以从文件中读取，也可以向文件中写入，如果文件存在，则会删除其中的数据。
- a+：打开文件，既可以从文件中读取，也可以向文件中写入。在新数据被写入文件之前会移除 EOF 标记。当写入操作完成时，会恢复 EOF 标记。如果文件不存在，则会创建新文件。

FILE 表示返回的打开的文件指针。如果返回值是 NULL 指针，则表示在打开文件时发生错误了。fopen()函数和_wfopen()函数的区别在于_wfopen()函数是 fopen()函数的多字符集版本。

3.10.2　从文件读取数据

从文件或数据流中读取数据使用 fread()函数，其原型为：

```
size t fread(
    void *buffer,                   //指定存储数据的本地存储空间的指针
    size_t size,                    //指定存储区的大小
    size_t count,                   //指定要读取的数据项的最大个数
    FILE *stream                    //指向 FILE 结构的指针，是使用 fopen()函数
                                    //打开的文件句柄
);
```

fread()函数会从文件流 stream 中读取 size 参数和 count 参数之间较小数目的数据，存储到 buffer 参数指定的缓冲区中。函数的返回值是实际读取的数据长度，当读取的过程发生错误或已经读到文件结尾处时，返回值可能小于参数 count 指定的值。使用 feof()函数和 ferror()函数可以区分这两种情况。如果参数 size 或参数 count 为 0，则函数返回 0，并且不会读取数据。

3.10.3　向文件写入数据

向文件中写数据使用 fwrite()函数，其原型为：

```
size t fwrite(
    const void *buffer,         //指定存放要写入的数据的存储空间的指针
    size_t size,                //指定存储区的大小
    size t count,               //指定要写入的数据项的最大个数
    FILE *stream                //指向 FILE 结构的指针，即使用 fopen()函数
                                //打开的文件句柄
);
```

fwrite()函数会将 buffer 参数指定缓冲区中的数据写入文件流 stream 中，写入的数据个数是 size 参数和 count 参数之间较小的数目。函数返回实际写入的数据长度，当写数据发生错误时，则返回值可能小于参数 count 指定的值。如果参数 size 或参数 count 为 0，则函数返回 0，并且不会向文件流中写任何数据。

3.10.4　关闭文件

C++中使用 fclose()函数或_fcloseall()函数关闭文件。其中 fclose()函数关闭文件流，_fcloseall()函数关闭所有打开的文件流。_fcloseall()函数还可以关闭和删除 tmpfile()函数创建的临时文件。当文件流关闭时，会释放系统分配的缓冲区。函数原型为：

```
int fclose( FILE *stream );          //指向 FILE 结构的指针
int _fcloseall( void );
```

如果成功关闭文件流，则 fclose()函数返回 0。_fcloseall()函数用于返回关闭的文件流的个数。当关闭文件流发生错误时，函数会返回 EOF。

3.10.5　文件操作示例

从上面讲解的内容可以看出文件操作的过程是这样的，先根据需求使用合适的权限打开文件，判断是否成功打开文件，并对文件进行读写，文件使用完后，需要关闭文件句柄。以下代码是文件操作的示例。

```
01  #include <stdio.h>
02  #include <process.h>
03
04  FILE *stream, *stream1, *stream2;
05
06  void main()
07  {
08      int numclosed;
09      char list[30];               //存放从文件中读取的数据
10      int  i, numread, numwritten;    //读取的数目，写入的数目
11
12      //打开文件 data 进行读，如果文件不存在，则失败
13      if( (stream1  = fopen( "data", "r" )) == NULL )
14          printf( "打开文件'data'进行读失败\n" );
15      else
16          printf( "打开文件'data'进行读\n" );
17
18      //打开文件 data2 进行写操作
19      if( (stream2 = fopen( "data2", "w+" )) == NULL )
20          printf( "打开文件'data2'进行写失败\n" );
21      else
22          printf( "打开文件'data2'进行写\n" );
23
24      //使用文本模式打开文件，对文件进行写操作
25      if( (stream = fopen( "fread.out", "w+t" )) != NULL )
26      {
27          //向文件流中写入 25 个字符
28          for ( i = 0; i < 25; i++ )
```

```
29                  list[i] = (char)('z' - i);
30
31              numwritten = fwrite( list, sizeof( char ), 25, stream );
32              printf( "写入 %d 个字符\n", numwritten );
33              fclose( stream );
34          }
35          else
36              printf( "打开文件 fread.out 时，发生错误，无法写数据到文件中\n" );
37
38          if( (stream = fopen( "fread.out", "r+t" )) != NULL )
39          {
40              //从文件中读取 25 个字符
41              numread = fread( list, sizeof( char ), 25, stream );
42              printf( "读取的数据个数 = %d\n", numread );
43              printf( "读取的内容为 = %.25s\n", list );
44              fclose( stream );
45          }
46          else
47              printf( "打开文件 fread.out 时，发生错误，无法从文件中读取数据\n" );
48
49          //关闭文件
50          if( fclose( stream2 ) )
51              printf( "关闭文件'data2'失败\n" );
52
53          //关闭其他打开的文件
54          numclosed =  fcloseall( );
55          printf( "使用函数_fcloseall 关闭的文件数目为 : %u\n", numclosed );
56          system("pause");
57  }
```

上面代码演示了操作文件的方法，包括打开文件、读文件、写文件和关闭文件。其中，对文件句柄 stream1 和 stream2 没有进行读写操作，对文件句柄 stream 进行了读写操作。首先向文件中写入 25 个字母，然后再从文件中读取这 25 个字母，最后关闭文件。代码运行结果如图 3-12 所示。

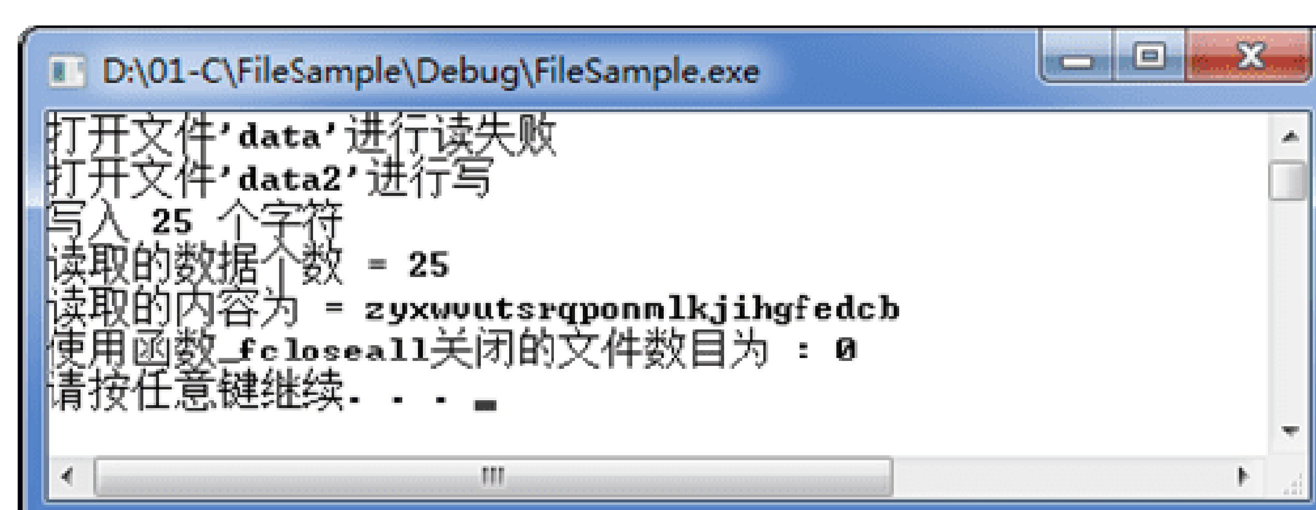


图 3-12　代码运行结果

3.11　本 章 小 结

本章主要讲解了 C++的基本语法，在与 C 语言语法对比的基础上，讲述了 C++语言特有的特性。从基本的变量和常量，到控制语句的语法以及函数的概念，C++在继承 C 的基础上做了适当调整。C++中的指针和引用是 C++语法中的一个难点。预处理的理解也是熟悉 C++语言必不可少的。最后以文件操作为例，讲述了在 C++中操作文件的方法，并结合实例进行说明。第 4 章将讲解 C++语言中的面向对象程序设计的知识。

3.12　习　　题

1．定义一个结构体 Person，用来记录人的姓名、性别、年龄、身份证号码和年收入。

【思路】依据结构体各成员存放的数据的情况，来决定所使用的数据类型。

【示例代码】

```
struct Person
{
    char name[9];
    char sex[3];
    short age;
    char identityCard[19];
    float income;
};
```

2．有一个式子：$(x + 303 * y - 64)/z$ 。x、y、z 分别为 int 型的变量，编程实现：从用户处获取各个变量的值，默认赋值 1，代入表达式中计算并输出计算结果。

【思路】通过 cin 获取用户的输入，代入表达式中求值，用 cout 输出。如图 3-13 所示，为编程实现效果的一种方式。

3．声明一个 short 类型的数组 Num，包含 10 个数据成员，成员依条件赋值，条件是：3*n+1（n 是大于 0 小于 11 的整数）。然后通过指针 pNum 来修改数组中下标为偶数的成员，统一设置为 0，最后输出所有的数组成员。

【思路】按规则为数组赋值，然后用指针指向数组并按照规则修改数组成员，最后循环输出所有的数组成员。如图 3-14 所示为演示的一种输出效果。

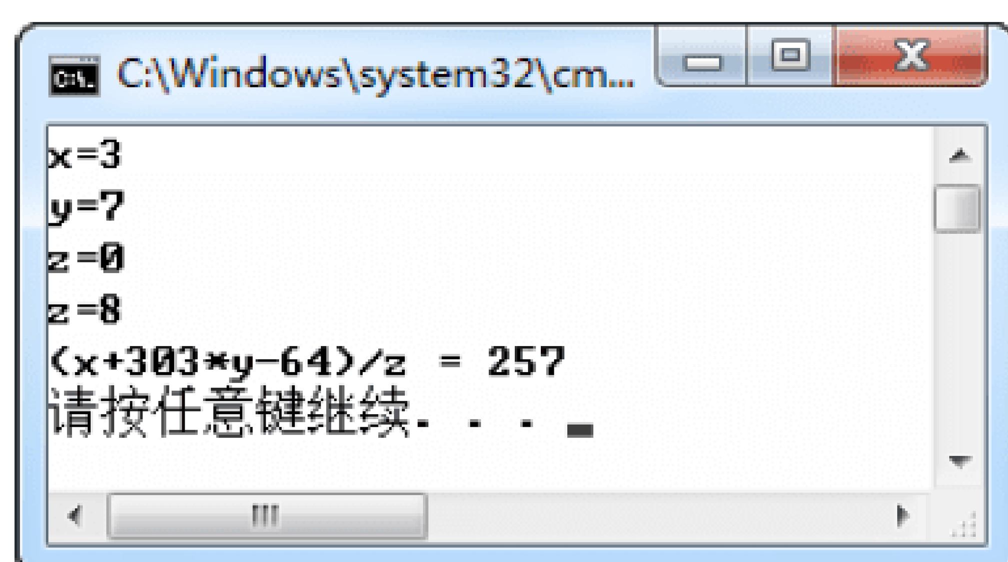


图 3-13　程序运行效果

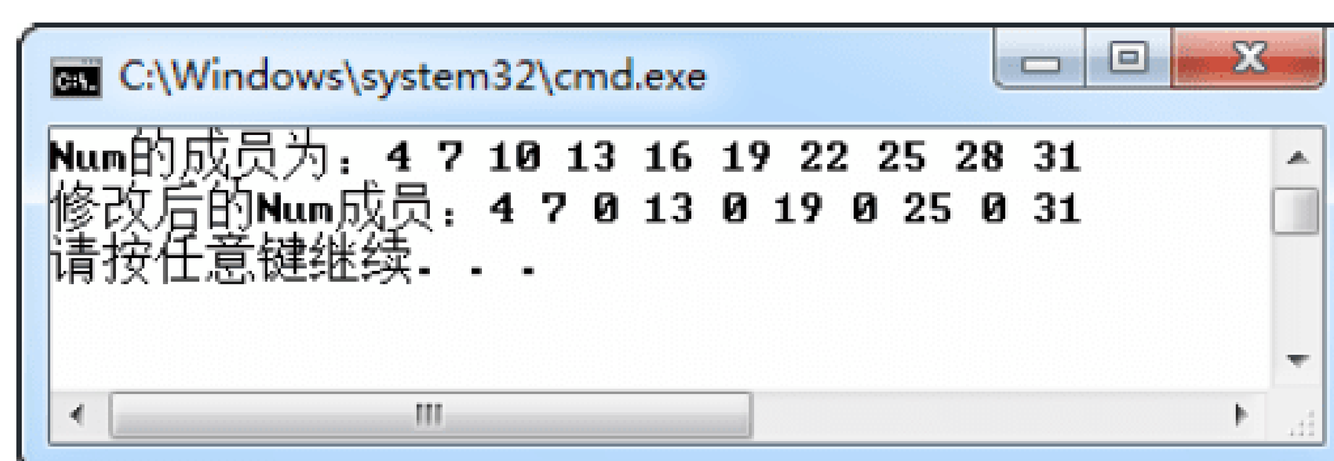


图 3-14　数组输出效果

4．依据用户输入的完整文件名（包含路径），打开并读取文本文件内容，然后显示出来。

【思路】使用 fopen()打开文件、fread()读取文件、feof()判断是否到达文件末尾。如图 3-15 所示为读取文件内容的效果。

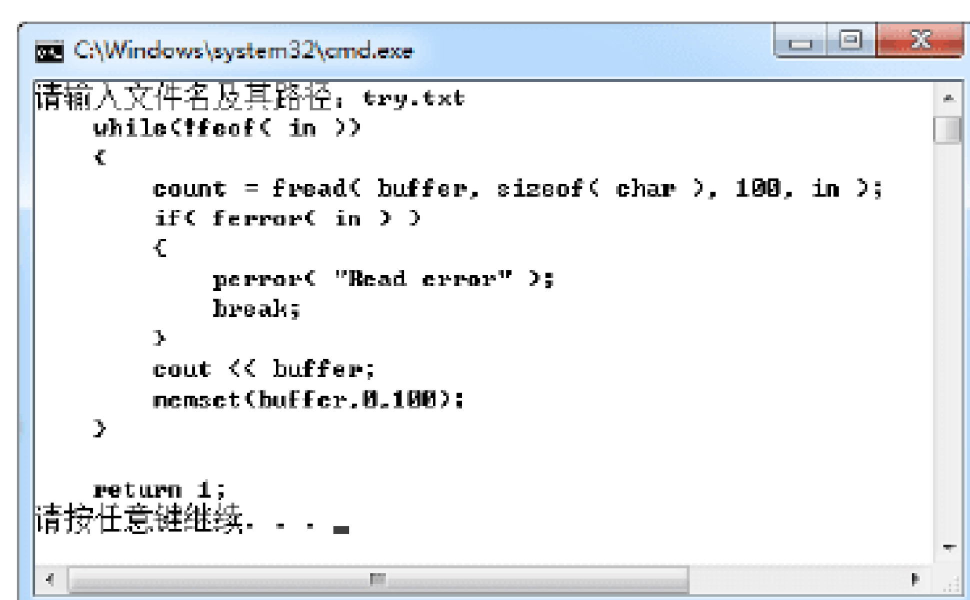


图 3-15　读取文件内容效果

第 4 章　C++面向对象程序设计

C++是在 C 基础上发展而来的面向对象的编程语言，它与 C 语言最大的区别就是，C 是面向过程的，而 C++是面向对象的。因此学习 C++语言的核心就是掌握面向对象的程序设计思路。本章将介绍 C++的面向对象程序设计思想。

4.1　类 和 对 象

类是面向对象程序设计中非常重要的概念，它可以包括数据和函数。C++使用类向对象中引入用户自定义类型。类的实例，称为对象，即对象是类类型的变量。在 C++中，复杂的数据类型一般使用类来实现。本节将介绍有关类和对象的基本概念。

4.1.1　从结构到类

传统的编程语言中用户自定义类型就是数据的集合，用于描述对象的属性和状态，但是不能描述对象的行为。为了解决这个问题，C++中引入了类的概念，不仅可以描述对象的属性和状态，还可以定义对象的行为。C++中使用 class、struct 和 union 3 个关键字定义类的类型，分别表示类、结构和共用体，虽然这 3 种类型都表示类，但是它们之间又存在区别，如表 4-1 所示。

表 4-1　结构、类和共用体的区别

结　　构	类	共　用　体
使用 struct 定义	使用 class 定义	使用 union 定义
默认的访问权限是公开的	默认的访问权限是私有的	默认的访问权限是公开的
没有使用限制	没有使用限制	同一时间只能使用一个成员
只描述属性和状态	描述属性、状态和对象行为	只描述属性和状态

其中，结构和共用体在传统的 C 语言中就支持，是用户自定义类型的数据集合，组合起来描述对象的属性和状态。C++类不仅可以使用户描述属性和状态，还可以定义行为。C++类是包含数据成员和函数成员的类型，使用类可以将用户自定义类型引入程序。

4.1.2　定义类

类是数据成员、操作这些数据成员的函数以及指定数据成员和函数的访问控制的组合。类的变量和函数称为类的成员。默认情况下，类成员是私有的，并且是私有继承的。

通常一个类由下面的成员组成。

- ❑ 类的数据成员：定义类对象的状态和属性。
- ❑ 一个或多个构造函数：用于初始化类对象。
- ❑ 一个析构函数：用于完成清除工作，如释放动态分配的内存、关闭文件等。
- ❑ 类的成员函数：用于定义对象的行为。

类定义语法规则为：

```
类关键字（class 或 struct 或 union） __declspec（可选） 类名（可选） 基类名（可选）
{
    类的成员列表
}
```

在声明类后定义类前，编译器就将类名作为标识符。因此，类是可以嵌套的。例如，可以将 Tree 类的 Left 成员和 Right 成员都定义为 Tree 类的类型。代码如下：

```
class Tree                  //定义表示二叉树的类
{
public:
    void *Data;             //结点数据
    Tree *Left;             //左子结点
    Tree *Right;            //右子结点
};
```

此外，可以使用 typedef 隐藏类名，代码如下：

```
typedef struct              //定义表示点的结构
{
    unsigned x;             //表示点的 X 坐标
    unsigned y;             //表示点的 Y 坐标
} POINT;
```

此种用法一般用于实现与已有 C 代码之间的兼容。在 C 代码中，将 typedef 与匿名结构一起使用是常用的方法。匿名定义类还有一种用法，就是直接将匿名类定义在另一个类中。代码如下：

```
struct PTValue              //定义存储点值的结构
{
    POINT ptLoc;            //定义 POINT 类型的变量
    Union                   //定义存储值的共用体
    {
        int  iValue;        //定义整型值
        long lValue;        //定义长整型值
    };
};
PTValue ptv;                //定义 PTValue 类型的变量
```

在上面代码中，iValue 成员可以使用对象成员选择符访问，例如：

```
int i = ptv.iValue;         //获取 PTValue 类型的 iValue 的成员值，并存入整型变量 i 中
```

类定义完成的地方是类的定义点，并且类的成员函数的定义是没有先后顺序的。代码如下：

```
class Point                         //定义表示点的 Point 类
```

```
{
public:
    Point() { cx = cy = 0; }//定义不带参数的构造函数
    //定义带参数的构造函数，传入的参数是点代表的坐标值
   Point( int x, int y )  { cx = x, cy = y; }
   unsigned &x( unsigned );//横坐标访问器
   unsigned &y( unsigned );//纵坐标访问器
private:
   unsigned  cx, cy;        //存储位置点的 X 值和 Y 值
};
```

在上面代码中，虽然 x 和 y 两个访问函数没有定义，但是类 Point 认为这两个函数已经定义了。

4.1.3　定义对象

类对象使用类名定义。代码如下：

```
class Account                //Account 类
{
public:
      Account();             //默认的构造函数
      Account( double );     //带参数的构造函数
}:
Account chkAccount;          //定义 Account 类的对象，对象名为 chkAccount
```

上面代码首先声明了名为 Account 的类，然后定义了对象名为 chkAccount 的 Account 类对象。

C++中有一种特殊的类——空类，虽然是空类，但是由空类定义的对象的大小不是 0。代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  class NoMemClass                //定义不包含任何成员的 NoMemClass 类
05  {
06  };
07  void main()
08  {
09      NoMemClass  obj;            //NoMemClass 类的对象
10      //输出类空类占用的空间大小
11      cout << "空类对象的大小为: " << sizeof(obj)  << endl;
12      system("pause");
13  }
```

上面代码定义了空类 NoMemClass，长度为 1，程序运行结果如图 4-1 所示。

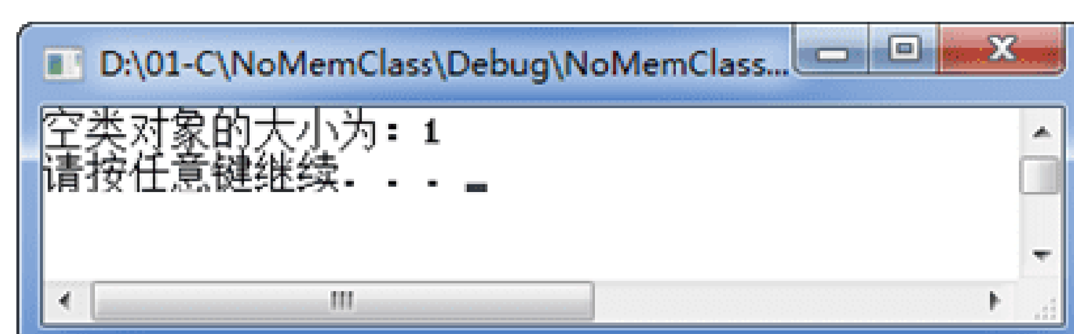


图 4-1　空类大小测试示例运行结果

4.1.4　嵌套类

在 C++中，类可以在其他类的范围内声明，这样的类称为嵌套类。嵌套类的声明与类的声明相同，只是声明的位置是在其他类的范围内。代码如下：

```
class BufferIO                                  //定义输入输出缓冲区类
{
public:
    enum IOError { None, Access, General };//定义表示输入输出错误的枚举值
    class BufferInput                           //定义嵌套输入缓冲区类
    {
    public:
        int read();                             //读取数据
        int good() { return  inputerror == None; }//判断是否有错误
    private:
        IOError _inputerror;                    //存储当前的错误值
    };
    class BufferOutput                          //定义嵌套类 BufferOutput
    {
        //成员列表
    };
};
```

上面代码在 BufferIO 类中定义了 BufferIO::BufferInput 类和 BufferIO::BufferOutput 类。因此，这两个类只在 BufferIO 类的范围内有效。注意，上面的代码中，BufferIO 的对象不包含 BufferInput 和 BufferOutput 类型的对象，只是声明了这两个类，并没有定义这两种类型的对象。在嵌套类定义的类中定义的变量和类型，在嵌套类中可以使用。如在 BufferIO::BufferInput 类中可以使用在 BufferIO 类中定义的 IOError 枚举类型。

嵌套类只在定义的类的范围内有效。要引用一个嵌套类，则必须指定完整的类名，即在使用此类时，要指定所属的类名。如在类外引用 BufferIO::BufferInput 类的 read()方法，代码如下：

```
int count = BufferIO::BufferInput::read();
```

在实际编写程序时，当类之间有主从所属关系时，使用嵌套类可以使对象的层次更加清晰。

4.2　类成员及其特性

类是 C++引入的新类型，此类型包含特有的成员。要想充分利用类的优点，需要了解类成员的使用。如类的构造函数和析构函数是类特有的成员。本节将介绍类的成员及其特性。

4.2.1　构造函数

C++中与类名称相同的成员函数称为构造函数，构造函数没有返回值。如果为类指定

了构造函数，则此类型的对象会在创建时使用构造函数进行初始化。即使没有在构造函数中写任何代码，构造函数也会执行默认的操作，完成必要的初始化工作。在 VC 中构造函数会依次执行下列任务。

（1）如果类是从虚基类中继承而来，则首先初始化对象的虚基类指针。

（2）按照声明的顺序，调用基类的构造函数。

（3）如果类有虚函数或继承虚函数，则初始化对象的虚函数指针。虚函数指针指向类的虚函数表（v-table），因为虚函数是“晚绑定”，所以使用虚函数表可以重新绑定虚函数调用的代码。

（4）执行构造函数体中的代码。

当执行构造函数时，系统会为类对象分配内存。构造函数还会构造基类和组合对象。同一个类可以根据需要声明多个构造函数，其语法格式如下：

```
类名 ( 可选的参数声明 ) 模式
```

C++定义了两种特殊的构造函数——默认构造函数和复制构造函数。默认的构造函数既可以声明为无参数的默认构造函数，也可以根据需要声明一个带参数列表的默认构造函数。带参数列表的默认构造函数提供所有默认的参数，作用是构造类的默认对象；复制构造函数具有一个与类有相同类型的引用作为参数。如果没有定义默认构造函数或复制构造函数，则编译器会自动生成一个默认构造函数和复制构造函数。

除了在创建对象时调用构造函数外，还可以显式地调用构造函数。代码如下：

```
DrawLine( Point( 0, 0 ), Point( 99, 99 ) );
Point pt = Point( 25, 50 );
```

在上面的代码中，第一条语句创建了两个 Point 对象，传入 DrawLine()函数中，并在函数退出时，析构这两个对象。第二条语句使用带有两个 int 类型参数的构造函数创建一个 Point 对象并对其进行初始化。

通常情况下，在构造函数中可以调用任何成员函数，包括虚函数。因为对象在执行用户代码的第一行时就已经完成了初始化工作。但是在抽象基类的构造函数和析构函数中不能调用虚成员函数。派生类型的对象的构造函数的执行顺序，是按照顺序依次调用从基类到派生类的构造函数。每个类的构造函数依赖于基类完成构造。包含类成员数据的类称为“组合类”。当创建组合类对象时，在调用构造函数前，首先执行被包含的类的构造函数。以下代码说明了构造函数的执行顺序。

```
01  #include <iostream>
02  using namespace std;
03
04  class Base                                   //定义基类
05  {
06  public:
07      Base();                                  //基类的默认的构造函数
08      virtual void f();                        //虚成员函数
09  };
10  Base::Base()                                 //基类的构造函数
11  {
12      cout << "构造 Base 对象\n";              //输出提示信息
13      f();                                     //在构造函数内调用虚成员函数
14  }
```

```
15  void Base::f()                          //定义 Base 类的 f()函数
16  {
17      cout << "调用了 Called Base::f()\n"; //输出提示信息
18  }
19  class Derived : public Base             //定义派生自 Base 的派生类
20  {
21  public:
22      Derived();                          //派生类的默认的构造函数
23      void f();                           //实现派生类的虚函数
24  };
25  Derived::Derived()                      //定义 Derived 类的构造函数
26  {
27      cout << "构造派生对象\n";            //输出提示信息
28  }
29  void Derived::f()                       //定义 Derived 类的 f()函数
30  {
31      cout << "调用 Derived::f()\n";       //输出提示信息
32  }
33  void main()                             //主函数
34  {
35      Derived d;                          //定义派生类 Derived 的变量 d
36  }
```

当上面的程序运行时，Derived d 调用下列事件：

（1）调用类 Derived 的构造函数。

（2）进入 Derived 类的构造函数之前，先调用 Base 类的构造函数。

（3）Base 的构造函数调用 f()函数，该函数是一个虚函数。通常 Derived::f()函数会被调用，因为对象 d 是 Derived 类型的。因为 Base::Base()函数是构造函数，对象还不是 Derived 类型，因此会调用 Base::f()函数。

注意：在数组中，对象只使用默认的构造函数进行构造，不使用任何参数，或所有参数都有默认值，并且数组总是按照升序构造数据。数组中每个成员的初始化使用相同的构造函数完成。

4.2.2　析构函数

析构函数是构造函数的逆操作。当对象销毁或释放时，系统会调用析构函数。指定析构函数的方法是在类中增加一个函数，此函数的函数名为类名前加一个~符号。离开对象作用域或用户调用对象的析构函数时，析构函数会清除对象所占用的资源。下面的代码是 Point 类的定义：

```
01  class Point                             //定义表示点的 Point 类
02  {
03      Point(char *str);                   //定义带参数的构造函数
04      ~Point();                           //声明析构函数
05  private:
06      double x;                           //定义 X 坐标值
07      double y;                           //定义 Y 坐标值
08      char*  desc;                        //位置描述变量
09  }
```

```
10   Point::Point( char *str )                 //定义构造函数
11   {
12       //动态分配位置描述成员所需要的存储空间
13       desc = new char[strlen( str ) + 1];
14       //如果分配成功，则将使用传入的数据初始化
15       if( desc )   strcpy( desc, str );
16   }
17   Point::~Point()                           //定义析构函数
18   {
19       delete[] desc;                        //释放位置描述成员所占用的空间
20   }
```

上面代码定义了 Point 类，其中 Point()函数是构造函数，~Point()函数是析构函数。在析构函数中释放动态分配给 desc 成员的存储空间。

4.2.3　对象成员初始化

构造函数使用类对象的构造函数初始化对象。默认的初始化对象的方法是从准备初始化的对象中按位依次复制。如下几种情况。

- 内建类型对象：以下代码为整型 i 初始化值 100。

```
int i = 100;
```

- 指针：以下代码为指针类型赋值为指向整型 a 的地址。

```
int a;
int *pa = &a;
```

- 引用：以下代码将引用类型 log 指定为 sFile 变量的引用。

```
String sFile( "20090101.log" );
String &log = sFile ;
```

- 类对象：当类对象没有私有成员、受保护成员、虚函数和基类时，才可以使用此种方式初始化。代码如下：

```
struct Point                            //定义 Point 结构
{
   double x, y;                         //定义 double 类型的坐标值
};
Point pt = { 120.3564, 36.123 };      //只有静态存储的类
```

除了默认的构造函数，还可以为类定义更多的构造函数。如果类中定义了构造函数，系统将不会再添加默认的构造函数。用户必须按照新定义的构造函数的方式初始化对象，否则，对象将无法初始化。并且类在实例化为对象时，只能执行一个构造函数。

4.2.4　常类型（const）

C++中使用 const 关键字定义常类型。其语法格式为：

```
const 声明
成员函数 const
```

在上面的语句中，第一条语句使用 const 关键字定义常对象，其值在程序中是不可以修改的。第二条语句使用 const 关键字定义常函数，即只读函数。它不可以修改所在对象中的任何数据成员，也不能调用任何不是常类型的成员函数。当使用 const 关键字定义常类型时，需要在声明和定义中都加上 const 关键字。示例代码如下：

```
01  class Date                                  //常函数例子
02  {
03  public:
04     Date( int mn,);                          //构造函数
05     int getMonth() const;                    //只读函数，获取月值
06     void setMonth( int mn );                 //可写函数，写入月值
07  private:
08     int month;                               //存储月值的变量
09  };
10  int Date::getMonth() const                  //只读函数，获取月值
11  {
12     return month;                            //不能在函数中修改任何数据
13  }
14  void Date::setMonth( int mn )               //可写函数，写入月值
15  {
16     month = mn;                              // 修改数据成员
17  }
```

上面代码定义了表示日期的 Date 类，其中定义了 getMonth()常函数，此函数返回类中表示月份 month 变量的值，但是不修改任何类对象的数据成员，而定义的 setMonth()函数可以修改 month 变量的值。

4.2.5　使用 this 指针指向对象

C++为类、结构和共用体类型提供了只能在成员函数中使用的 this 指针，它指向调用成员函数的对象。只有非静态成员函数才可以使用 this 指针。当对象调用非静态成员函数时，对象地址会作为函数的隐藏参数传入。代码如下：

```
myPoint.setX( 120.54 );
myPoint.setX( & myPoint , 120.54 );
```

在上面的代码中，第一条语句会被编译为第二条语句的形式。在成员函数中，可以使用“this->成员名称”选择正确的函数或数据成员，也可以不使用 this 指针。要注意的是，C++中 this 指针是只读的，程序不可以为其赋值。表达式*this 通常用于从成员函数中返回调用函数的当前对象。以下代码中 3 条语句的作用是相同的。

```
01  void Point::setX( double inputX )   //this 指针的用法
02  {
03     x = inputX;                       //赋值 x 为 inputX 参数值
04     this->x = inputX;                 //赋值 x 为 inputX 参数值
05     (*this). x = inputX;              //赋值 x 为 inputX 参数值
06  }
```

4.2.6　类的作用域和对象的生存期

类的作用域是指定义的有效范围，类的数据成员和函数成员的作用域在类对象中。而类中定义的子类和在类范围内重命名的类型，其作用域也在类对象中，在类的外部是不能使用的。代码如下：

```
01  class Tree                    //定义存放二叉树的 Tree 类
02  {
03  public:
04     typedef Tree * PTREE;      //定义存放树的指针
05     PTREE  Left;               //指向左二叉树
06     PTREE  Right;              //指向右二叉树
07     void  *vData;              //树的数据变量
08  };
09  PTREE pTree;                  //发生错误，因为不在类范围内
```

从上面的代码中可以看出，PTREE 重命名了类型 Tree *，但是因为它是在类 Tree 内部定义的，所以，在类外部使用 PTREE 会导致错误，即上面代码的最后一行所示。

4.2.7　使用静态成员保存类的数据

类中可以包含静态数据成员和静态函数成员，统称为静态成员。默认情况下，静态成员的作用域是类范围内且是外部链接的。静态成员在内存中只有一个副本，不是类对象的一部分，而是独立的对象。因此，静态数据成员的声明不是定义，声明在类作用域内完成，而定义在文件作用域内完成。代码如下：

```
01  class BufferedOutput          //输出缓冲区类
02  {
03  public:
04      short BytesWritten()
05      { return bytecount; }     //返回写入类对象的字节数
06      static void ResetCount()
07      { bytecount = 0; }        //重置计数器
08     static long bytecount;     //静态成员，当前写入的字节数
09  };
10  //定义文件范围内的 bytecount 变量
11  long BufferedOutput::bytecount = 8;
```

在上面代码中，bytecount 静态成员在类 BufferedOutput 类中声明，但是必须在类声明外定义。静态成员的类型与类名无关，因此，BufferedOutput::bytecount 的类型是 long。如果使用 BufferedOutput 对象写入的字节数，可以使用以下代码获取：

```
long nBytes = BufferedOutput::bytecount;//获取输出缓冲区的 bytecount 变量值
```

因为静态成员的存在不依赖于对象实例，因此可以使用成员选择符（.和->）访问，也可以不使用类对象访问。代码如下：

```
01  BufferedOutput Console;                 //定义输出缓冲区变量
02  long nBytes = Console.bytecount;        //获取其中的数据长度
```

```
03  class WindowManager                         //对话框管理类
04  {
05  public:
06       static int  CountOf();                 //返回打开的对话框数目
07       void Minimize();                       //最小化当前对话框
08       WindowManager SideEffects();           //SideEffects()函数
09      ...                                     //此处代码省略
10  private:
11      static int wmWindowCount;               //对话框数目
12  };
13  int WindowManager::wmWindowCount = 0;       //为对话框管理类的静态成员赋值
14  ...                                         //此处代码省略
15  for( int i = 0; i < WindowManager::CountOf(); ++i )
16  {
17       //最小化所有的对话框
18       rgwmWin[i].Minimize();
19  }
```

在上面代码中，对象 Console 的引用不会处理，返回值是静态对象 bytecount。类 WindowManager 包含 CountOf()静态成员函数，此函数返回打开的对话框的数目，调用此函数不需要引用 WindowManager 对象，而直接使用 WindowManager 访问即可。

静态数据成员的访问也是遵循类成员访问规则的。私有静态数据成员只允许类成员函数和友元访问。与非静态成员函数相比，静态成员函数没有 this 参数。因此，要注意以下几点。

- 静态成员函数不能使用成员选择操作符访问非静态成员数据。
- 静态成员函数不能声明为虚函数。
- 静态成员函数不能声明为与非静态函数具有相同参数类型的函数名。

4.2.8　友元函数和友元类

默认情况下，类的私有成员不允许其他类访问，类的受保护成员不允许继承类外的其他类访问。为了使指定类可以访问这些成员，C++提供了友元，允许函数或类访问类中的私有成员和受保护成员。在 C++中，使用 friend 来定义友元，友元必须在结构或类中定义。

1．友元函数

定义友元的语法是在 friend 加上类名或函数声明，语法格式如下：

```
class XXX
{
    friend 函数声明;
    friend 类名;
}
```

在类定义中，第一条语句是定义友元函数，其中函数声明表示可以访问 XXX 中的私有成员的函数的声明。可以是全局函数，也可以是类的成员函数声明。代码如下：

```
01  class Point                               //友元函数示例，定义存储位置的 Point 类
02  {
03  public:
04       Point( double inputX, double inputY );      //带初始化参数的构造函数
```

```
05      friend Point Average( Point a, Point b );  //定义平均值的友元函数
06      double GetX();                             //获取 X 坐标
07      double GetY();                             //获取 Y 坐标
08  private:
09      double x, y;                                   //定义 x、y 坐标
10  };
11  Point :: Point( double inputX, double inputY ) //构造函数
12  {
13      x = inputX;                                //使用参数初始化 x 值
14      y = inputY;                                //使用参数初始化 y 值
15  }
16  Point Average( Point a, Point b )              //计算中间点的坐标
17  {
18      //返回两个点之间的中间点的坐标
19      return Point( (a.x + b.x)/2, (a.y + b.y)/2 );
20  }
21  double Point :: GetX()                         //获取 x 坐标值
22  {
23      return x;
24  }
25  double Point :: GetY()                         //获取 y 坐标值
26  {
27      return y;
28  }
29  void printFriendFunction()                     //输出友元类用例数据
30  {
31      //定义 Point 类型的变量 pointA
32      Point pointA((double)120.4, (double)36.1);
33      //定义 Point 类型的变量 pointB
34      Point pointB((double)120.1, (double)36.4);
35      Point pointAv = Average(pointA, pointB);   //计算平均值
36      //输出 pointA 坐标值、pointB 坐标值和中间点的坐标值
37      cout << "pointA(x,y)=" << "(" << pointA.GetX() << ","
38          << pointA.GetY()    << ")\n";
39      cout << "pointB(x,y)=" << "(" << pointB.GetX() << ","
40          << pointB.GetY()    << ")\n";
41      cout << "Average(x,y)=" << "(" << pointAv.GetX() << ","
42          << pointAv.GetY()   << ")\n";
43  }
```

上面代码定义了 Average()友元函数，用于计算两个点 x 的平均值和 y 的平均值。可以访问 Point 类的私有成员 x 和私有成员 y。

2．友元类

在类定义中，“friend 类名”表示友元类定义，其中类名表示可以访问 XXX 类中的私有成员的 XXX 类的名称。友元类的成员函数可以访问此类的所有的成员。代码如下：

```
01  class BabyClass                        //友元类示例
02  {
03      friend class MotherClass;          //声明友元类
04  private:
05      int money;                         //定义私有成员，存放宝宝压岁钱
06  public:
07      BabyClass();
08      int GetMoney();
09  };
```

```
10  BabyClass :: BabyClass()                    //构造函数
11  {
12      money = 0;
13  }
14  int BabyClass::GetMoney()                   //获取月值
15  {
16      return money;
17  }
18  class MotherClass                           //定义 MotherClass 类
19  {
20  public:
21      void change( BabyClass& bc, int money );  //声明 change()函数
22  };
23  void MotherClass::change( BabyClass& yc, int x )//定义 change()函数
24  {
25      //因为 Mother 类是 BabyClass 类的友元类，所以可以访问私有成员
26      yc.money += x;
27  }
28  void printFriendClass()                           //打印友元函数
29  {
30      BabyClass baby;                               //定义 BabyClass 类
31      MotherClass mother;                           //定义 MotherClass 类
32      //输出修改前的值
33      cout << "Mother 修改之前：money=" << baby.GetMoney() << "\n";
34      mother.change((BabyClass&)baby, 100);           //修改变量值
35      //输出修改后的值
36      cout << "Mother 修改之后：money=" << baby.GetMoney() << "\n";
37  }
```

上面的示例定义了两个类，即 MotherClass 和 BabyClass 类。BabyClass 类有个私有成员，用来存放压岁钱，其他类是不可以访问的。在 BabyClass 类中定义了 MotherClass 类是其友元类，即 MotherClass 类可以访问 BabyClass 类中所有的成员函数，包括私有成员函数和受保护成员函数。此外，MotherClass 类在 change()成员函数中，可以为 BabyClass 类增加私有成员 money 的值。其运行结果如下：

```
Mother 修改之前：money=0
Mother 修改之前：money=100
```

友元的继承和传递有特殊规定，因此在使用时需要注意以下几点。

- 友元不是双向的。如上例中，MotherClass 类是 BabyClass 类的友元类，但 BabyClass 类不是 MotherClass 类的友元类。因此，虽然 MotherClass 类可以访问 BabyClass 类的所有成员，但是 BabyClass 类只能访问 MotherClass 类的公共成员。如果要使友元双向成立，则必须在两个类中分别显式地定义其各自的友元类。
- 友元不具有继承性。如上例中，从 MotherClass 类中继承而来的类也不能访问 BabyClass 类的私有成员和受保护成员，除非显式地在 BabyClass 类中定义继承类为友元类。
- 友元不具有传递性。如上例中，MotherClass 类的友元类可以访问 MotherClass 类中的所有成员，不能访问 BabyClass 类的私有成员和受保护成员，但是 MotherClass 类的友元类可以访问 MotherClass 类中带有访问 BabyClass 类私有成员的成员函数。

4.3 继承与派生

继承是 C++语言非常重要的一个特点，指从已经存在的类中派生新类，使得新类可以复用父类定义的数据和函数。其中用于继承的类称为基类，继承而来的类称为派生类。本节将介绍有关继承和派生类的基本知识。

4.3.1 如何使用继承方法

继承的语法格式如下：

```
类名称定义: [[,][ [ private | protected | public] [virtual] ] | [ [virtual]
[private | protected | public] ] 基类名]+
```

上面列出了定义派生类的基本语法。首先是类名称定义，与前面介绍的类的定义相同。然后在类名称后加一个冒号。其后就是有关继承的定义。C++类可以在一个类中定义多个继承，每个继承定义之间使用逗号分隔开。每个继承由以下 3 部分组成。

- ❑ 一部分是访问修饰符，有 3 个关键字，即 private、protected 和 public，分别表示私有访问权限、受保护权限和公开保护权限。
- ❑ 一部分是 virtual 关键字，用于定义虚继承，在后面会详细介绍有关虚继承的内容。
- ❑ 一部分是基类名，指定了要继承的类的名称。

其中，访问修饰符部分和 virtual 关键字是可选的，并且这两部分的位置可以交换，但是基类名必须放在这两部分的后面。

根据继承的基类的数目，可以分为单一继承和多重继承。单一继承是继承的常用形式，其派生类只从一个基类派生而来，图 4-2 所示为单一继承的例子。

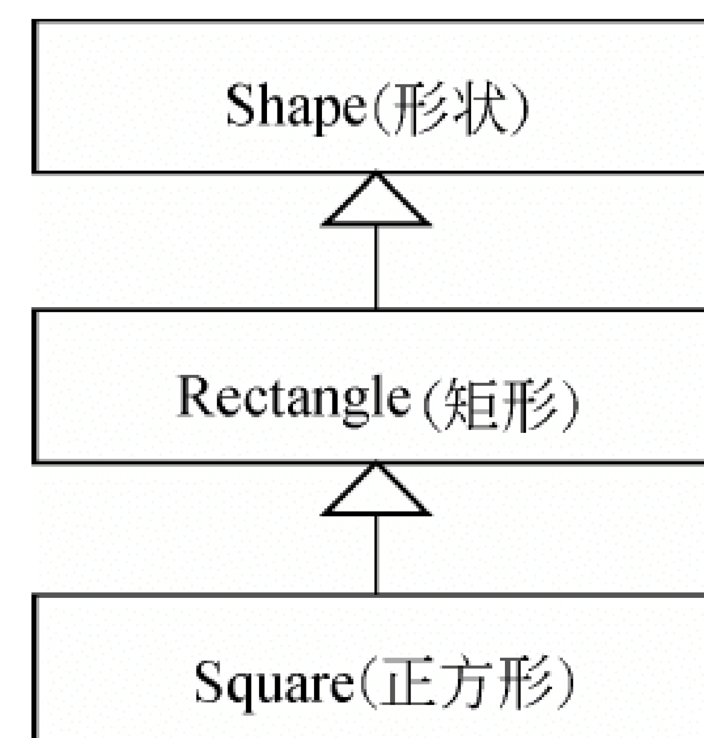


图 4-2 单一继承类示例图

从图 4-2 中可以看出，Rectangle 从 Shape 继承而来，Square 从 Rectangle 派生而来。三者之间具有种类的联系，即矩形是形状的一种，正方形是矩形的一种。因此，继承主要是处理类之间的种类关系。而其中的矩形既是从形状派生而来的派生类，同时也是派生正方形的基类。其声明代码如下：

```
01  class Shape                                     //定义形状类
02  {
03  public:
04      double  area;                               //形状的面积
05      double  CaluateArea();                      //计算面积
06      void    PrintArea();                        //输出面积
07  };
08  void Shape:: PrintArea ()                       //实现 Shape 类的 PrintArea()函数
09  {
10      cout << "当前没有形状设置" << endl;         //输出信息提示
11  }
12  double Shape::CaluateArea()
13  {
```

```
14      return area;
15  }
16  class Rectangle : public Shape          //Rectangle 从 Shape 派生
17  {
18  public:
19      Rectangle (double length, double width );
20      //矩形的构造函数，参数为长和宽
21  private:
22      double  Length;                     //长
23      double  Width;                      //宽
24  };
25  Rectangle :: Rectangle (double length, double width )
26      //实现矩形的构造函数
27  {
28      Length = length;                    //初始化长度
29      Width = width;                      //初始化宽度
30  };
31  void Rectangle :: CaluateArea ()        //实现矩形的构造函数
32  {
33      area = length*width;                //计算面积
34      Shape :: CaluateArea();             //调用基类的计算面积函数
35  };
36  class Square: public Rectangle          //Square 从 Rectangle 派生而来
37  {
38      //成员列表
39  };
```

其中，Shape 除了是 Rectangle 类的直接基类，还是 Square 类的间接基类。直接基类与间接基类的不同在于，直接基类出现在类声明的基类列表中，而间接基类则不出现在类声明的基类列表中。

在继承中，派生类包含基类的成员和新增加的成员。派生类可以引用基类的成员。使用范围确定符（::）可以指定引用的直接基类或间接基类的成员。上面代码中，CaluateArea()函数访问了数据成员 area。继承成员的使用方法和类成员的使用方法是相同的。当调用 Rectangle 类的 CaluateArea()重写函数时，要调用 Shape 的 CaluateArea()函数，则需要使用范围确定符（::）。

4.3.2　派生类的构造函数和析构函数

派生类的构造函数和析构函数是派生类中最需要注意的两个特殊函数，它们可以继承自基类，但是有固定的执行次序。构造函数首先执行基类的构造函数，再执行派生类的构造函数；而析构函数的执行顺序与构造函数的执行顺序相反，首先执行派生类的析构函数，再执行基类的析构函数。定义构造函数的语法格式为：

```
派生类名::派生类名（参数列表）: 基类名 1（参数列表）,基类名 2（参数列表）...
{
    //派生类的构造函数，在其中执行初始化功能
}
```

析构函数与构造函数的定义方式相同。以下代码显示了派生类的构造函数和析构函数的实现：

```
01  #include <iostream>
02  using namespace std;
03
04  class Shape                         //定义基类形状类
05  {
06  public:
07      Shape();                        //声明基类的构造函数
08      ~Shape();                       //声明基类的析构函数
09  };
10  Shape::Shape()                      //定义基类的构造函数
11  {
12      cout << "这是基类 Shape 的构造函数" << endl;
13  }
14  Shape::~Shape()                     //定义基类的析构函数
15  {
16      cout << "这是基类 Shape 的析构函数" << endl;
17  }
18  class Rectangle : public Shape      //Rectangle 从 Shape 派生而来
19  {
20  public:
21     Rectangle ();                    //声明派生类的构造函数
22      ~Rectangle();                   //声明派生类的析构函数
23  };
24  Rectangle::Rectangle()              //定义派生类的构造函数
25  {
26      cout << "这是派生类 Rectangle 的构造函数" << endl;
27  }
28  Rectangle::~Rectangle()             //定义派生类的析构函数
29  {
30      cout << "这是派生类 Rectangle 的析构函数" << endl;
31  }
32  void main()                 //测试派生类的构造函数和析构函数
33  {
34      Rectangle rect;
35  }
```

上面代码定义了基类 Shape 表示形状类，在构造函数和析构函数中向输出设备输出信息提示。然后定义了继承自 Shape 的派生类 Rectangle 表示矩形，在构造函数和析构函数中向输出设备输出信息提示。程序运行效果如图 4-3 所示。

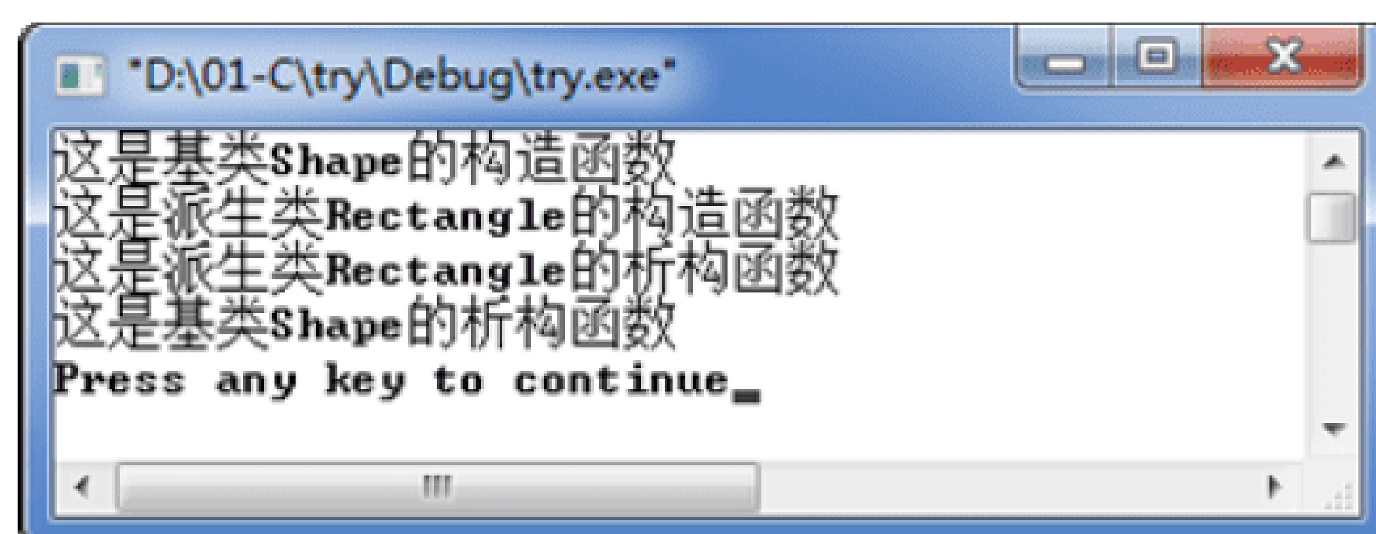


图 4-3　派生类的构造函数和析构函数程序运行效果

从图 4-3 的运行效果可以看出，对于构造函数，首先执行基类 Shape 的构造函数，再执行派生类 Rectangle 的构造函数；而析构函数执行时，首先执行派生类 Rectangle 的析构函数，再执行基类 Shape 的析构函数。

4.3.3　实现多重继承

C++中支持多重继承，即派生类派生自多个直接基类，如图 4-4 所示。

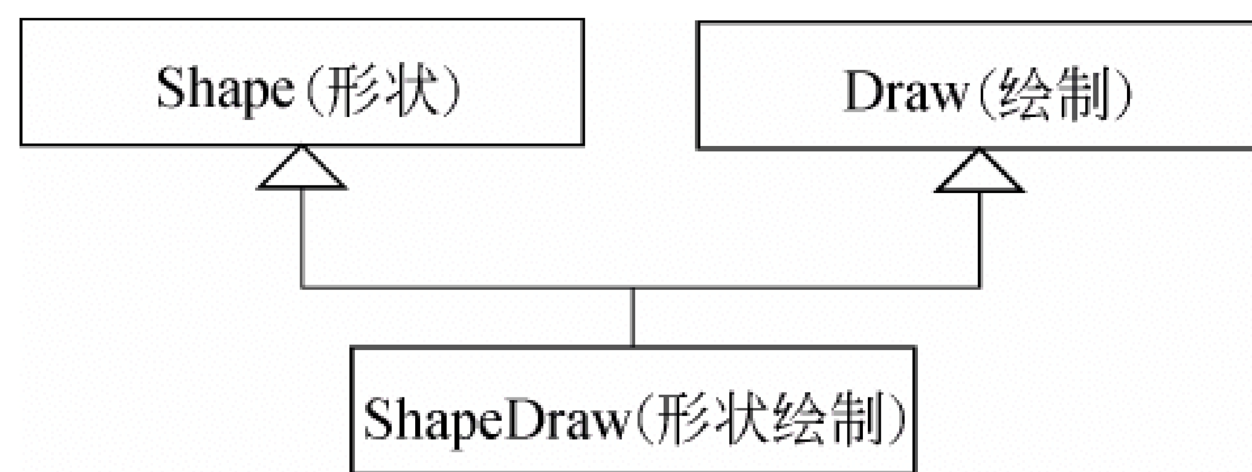


图 4-4　多重继承例子——矩形绘制类

图 4-4 所示中的矩形绘制类，作用是绘制形状，既是形状，又要进行绘制，因此，将 Shape 类和 Draw 类结合起来使用多重继承是最好的解决办法。之所以使用多重继承是为了将形状和绘制两个特性分开，创建其他使用其中特性的类时，可以使用相同的类继承，如图片的绘制、形状的填充等都可以使用其中的绘制和形状进行继承，而使用单一继承则无法实现这样的灵活应用。

多重继承的语法与继承语法相同，只是将多个基类列在基类列表中，各个基类间使用逗号分隔开，而且顺序没有特殊意义。在基类列表中，不能指定相同的类名，但是用于派生的多个基类可能派生自同一个间接基类。如下代码列出了继承自 Shape 类和 Draw 类的派生类 ShapeDraw 类的定义。

```
class ShapeDraw : public Shape, public Draw
{
    //新成员
};
```

虽然基类列表中各个基类的顺序没有特殊意义，但是其顺序会影响到下列处理。

- 构造函数执行初始化的顺序。当构造函数进行初始化时，会按照基类列表中各个基类的顺序执行基类的构造函数。
- 析构函数执行清理工作的顺序。当析构函数进行清理工作时，会按照与基类列表中各个基类相反的顺序执行基类的析构函数。
- 基类的顺序还会影响类的内存配置。所以设计程序时，不要有针对继承顺序的设计。

在多重继承时，对于指针和引用之间的转换需要特别注意，因为多重继承会导致名称继承有可能多于一条路径。因此，在使用基类的成员时，必须使路径唯一。在引用间接基类时，应该使用对象的完整引用路径。例如，假设类 D 从类 B 和类 C 多重继承而来，而类 B 和类 C 都继承自类 A，如图 4-5 所示。

从图 4-5 中可以看出，声明 D 类型的对象 d 后，使用地址操作符可以获取对象的基地址，如&d 指向对象 d 的地址。使用通过地址操作符获取的对象的地址可以获取直接基类，如(B*)&d 为 d 对象的 B 子对象，(C*)&d 为 d 对象的 C 子对象。但是使用通过地址操作符获取的对象的地址，获取间接基类时，存在混淆，如(A*)&d。因为对象 d 具有两个基类，都继承自间接基类 A，这种表达方式会存在两个对象，从而出现混乱。此时，就需要使用

通过地址操作符获取的对象的地址获取指定基类的间接基类，如(A *)(B *)&d 表示 d 对象的 B 子对象的基类 A 对象，(A *)(C *)&d 表示 d 对象的 C 子对象的基类 A 对象。代码如下：

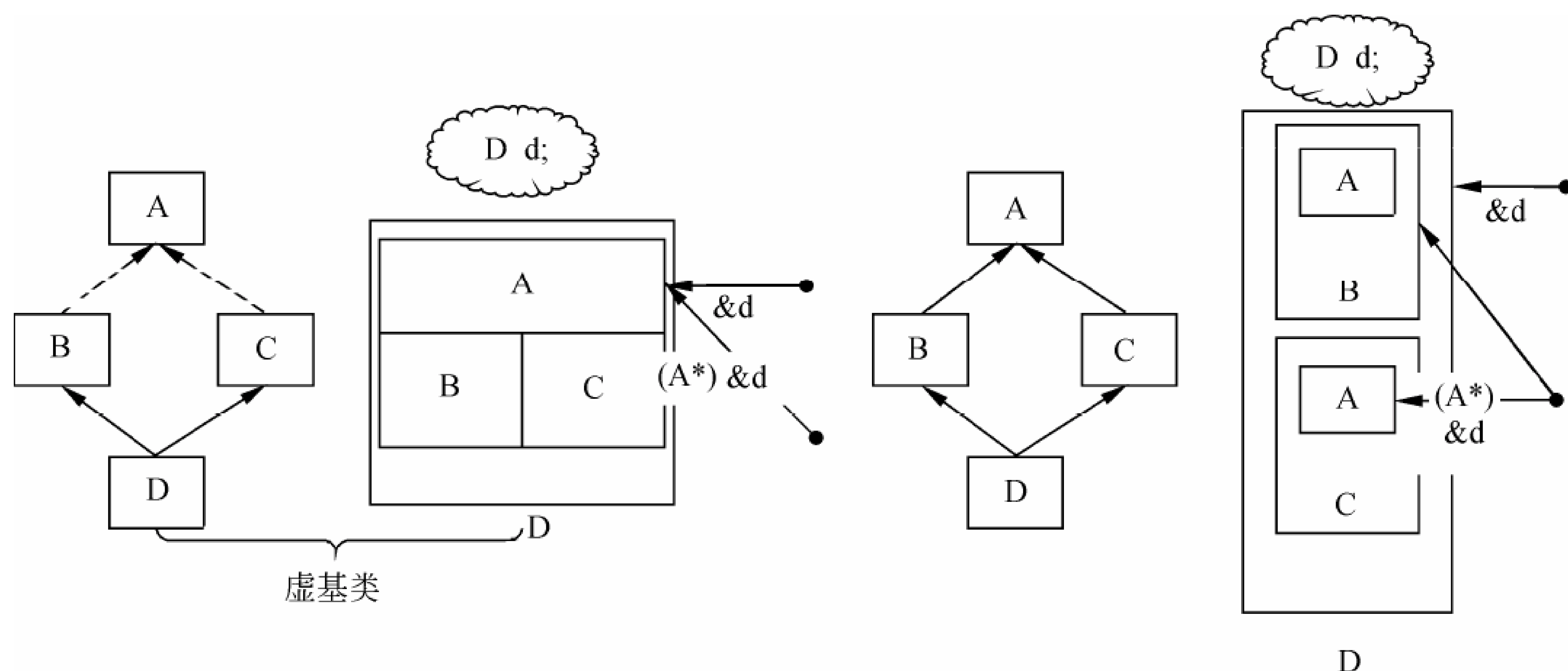


图 4-5　多重继承的指针引用

```
01  class A                             //声明两个基类——A 和 B
02  {
03  public:
04      unsigned a;                     //定义变量 a
05      unsigned b();                   //定义 b()函数
06  };
07  class B
08  {
09  public:
10      unsigned a();                   //定义 a()函数
11      int b();                        //定义 b()函数
12      char c;                         //定义成员 c
13  };
14  class C : public A, public B        //定义类 C 派生自 A 和 B
15  {
16  };
17  C *pc = new C;
18  pc->b();                 //错误：会导致不明确的调用
19                           //因为在 A 和 B 中都包含 b()函数成员
20  pc->B::b();              //正确：使用 B 的 b()函数
```

4.3.4　虚基类

因为派生类可能多次继承自同一个间接基类，所以 C++提供了一种优化此种基类的方法——虚基类。所有从虚基类中继承而来的派生类，会使用同一个基类对象，而每个从非虚基类中继承而来的派生类，会创建自己的基类对象，如图 4-6 所示。

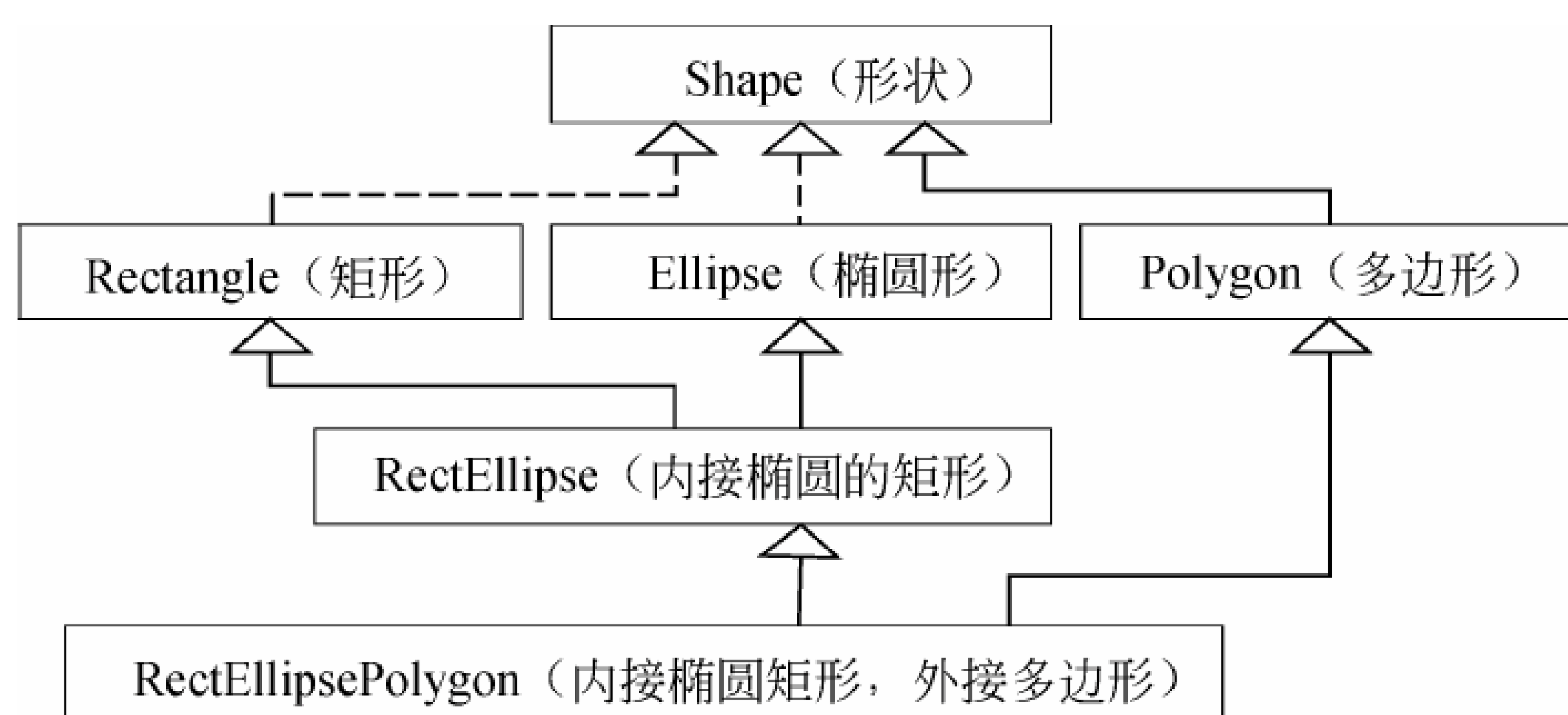


图 4-6　虚基类的实例

在图 4-6 中，Shape 是 Rectangle 类、Ellipse 类和 Polygon 类的基类，其中 Rectangle 类和 Ellipse 类是使用 Shape 类作为虚基类，Polygon 类是使用 Shape 类作为非虚基类。此情况下创建的类结构，如图 4-7 所示。

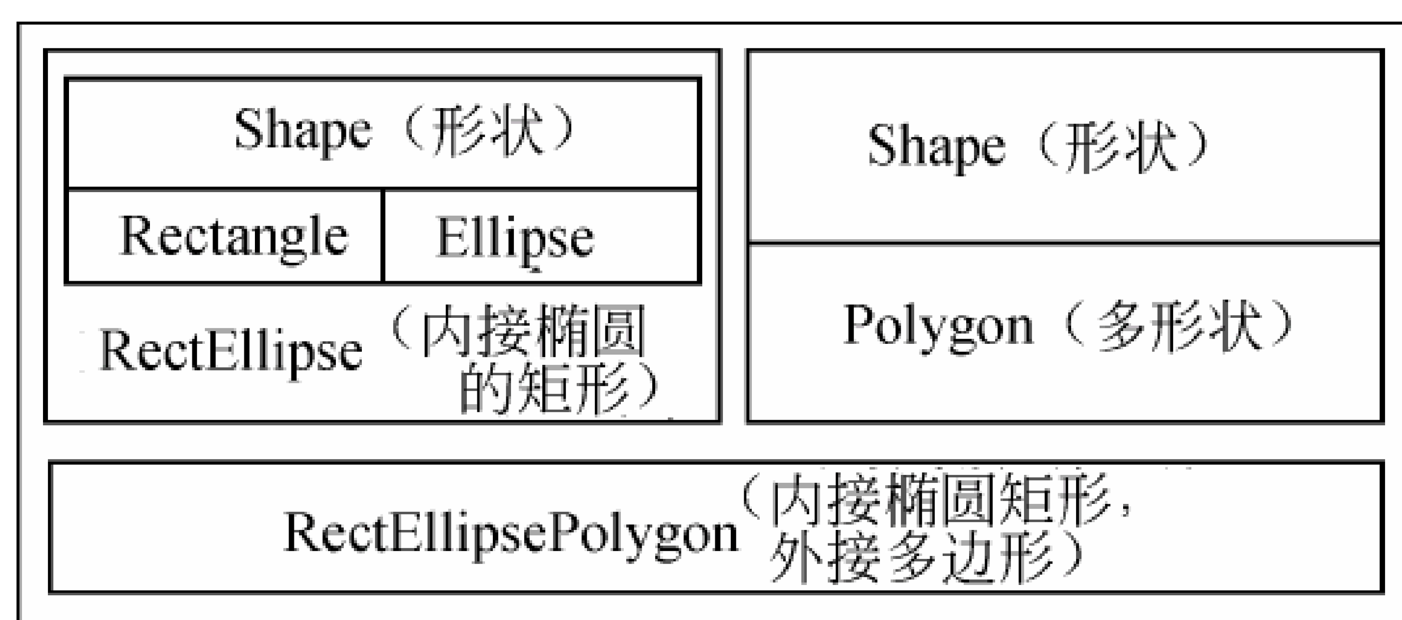


图 4-7　类结构图

从图 4-7 中可以看出，因为 Rectangle 类和 Ellipse 类使用虚基类 Shape，所以，这两个类共用同一个 Shape 基类的实例。而 Polygon 类使用非虚基类 Shape 继承，所以，它为自己创建基类对象。因为使用虚基类继承共用同一个基类，所以虚基类可以解决多重继承中引用不明确的问题。

4.4　多态和虚函数

虚函数是面向对象程序设计思想中比较高级的概念。因为面向对象程序设计思想的核心是将编程过程中操作的数据作为对象处理，而对象是对世界万物的抽象，因此有一个重要的概念就是虚函数。虚函数类似现实世界的“种类”。相同的种类具有相同的行为，具体行为的执行方式，根据个体不同而有差别。而虚函数是一类函数的抽象，这类函数具有相同的行为，具体函数的行为方式由它自己决定。

4.4.1　使用虚函数实现派生类的通用功能

虚函数的主要作用是将基类和各个派生类的通用功能抽象出来，实现通用部分，而在派生类中写特定代码。因此，从多个对象抽象出功能交集是非常重要的，而交集中的功能使用虚函数实现。有时基类会为这些虚函数提供一个默认的实现函数。

不管调用函数表达式是怎样的，虚函数都会根据规则调用正确的函数。如果基类声明了虚函数，而派生类定义了相同名称的函数，则不管使用基类的指针还是派生类的指针都会调用在派生类中定义的函数。以下代码修改了 4.3.1 小节中使用的形状例子。

```
01  class Shape                                 //定义形状类
02  {
03  public:
04      double  area;                           //形状的面积
05      virtual double   CaluateArea();         //计算面积
06      virtual void     PrintArea();           //输出面积
07  };
08  void Shape:: PrintArea ()                   //实现 Shape 类的 PrintArea()函数
09  {
10      cout << "当前没有形状设置" << endl;     //输出信息提示
11  }
12  double Shape::CaluateArea()
13  {
14      return area;
15  }
16  class Rectangle : public Shape              //Rectangle 从 Shape 派生
17  {
18  public:
19      //矩形的构造函数，参数为长和宽
20      Rectangle (double length, double width );
21  private:
22      double  Length;                         //长
23      double  Width;                          //宽
24  };
25  Rectangle :: Rectangle (double length, double width )
26  {
27      this->Length = length;                  //初始化长度
28      this->Width =  width;                   //初始化宽度
29  };
30  double  Rectangle :: CaluateArea()          //实现矩形的构造函数
31  {
32      area = length*width;                    //计算面积
33      Shape :: CaluateArea();                 //调用基类的计算面积函数
34  };
35  void Rectangle :: PrintArea ()              //实现矩形的 PrintArea
36  {
37      CaluateArea();                          //计算面积
38      cout << "矩形面积为（单位平方米）：" << area;  //输出面积
39  };
40  class Circle : public Shape
41  {
42  public:
43      Circle (int radio );
44  private:
45      int ra;
46  };
47  Circle::Circle(int radio)
48  {
49      ra = radio;
50  }
51  double  Circle::CaluateArea()
52  {
53      area = 3.14*ra*ra;
```

```
54      Shape :: CaluateArea();
55  }
56  void Circle :: PrintArea()          //实现 Circle 类的 PrintArea()函数
57  {
58      CaluateArea();                  //计算面积
59      cout << "圆形面积为（单位平方米）: " << area;  //输出面积
60  };
```

上面代码修改了派生类 Rectangle 类和 Circle 类的 PrintArea()虚函数，要调用此函数，代码如下：

```
01  //创建 Rectangle 类和 Circle 类的对象指针
02  Rectangle*pRect = new Rectangle ( 10, 20 );
03  Circle* pCir  = new Circle ( 10 );
04  Shape *pShape = pRect;                  //使用 Shape 指针调用 PrintArea
05  pShape->PrintArea ();                   //输出面积
06  pShape = pCir ;                         //使用 Shape 指针调用 PrintArea
07  pShape->PrintArea ();                   //输出面积
```

上面代码调用 PrintArea()的作用是相同的，不同之处在于其指向的对象不同。因为 PrintArea()是虚函数，分别会调用对象定义的函数版本，即在 Rectangle 类和 Circle 类中的 PrintArea()函数重写了基类 Shape 中的 PrintArea()函数。如果派生类中没有重写基类的虚函数，则当调用虚函数时，会执行基类的虚函数。

4.4.2　纯虚函数和抽象基类

纯虚函数是指不指定任何实现的函数。如果类只包含纯虚函数，或从纯虚函数继承而来，并且没有提供任何实现，则该类称为抽象类。程序中不能创建抽象类对象，只能使用抽象类进行派生，而所有从抽象类派生而来的类必须实现纯虚函数。纯虚函数使用纯指示符（=0）声明，代码如下：

```
virtual function1() = 0;
```

例如在形状类中，Shape 类提供了通用的功能，但是因为太通用以至于无法实现具体的功能。因此，Shape 类使用抽象类是比较好的设计。改写后的代码如下：

```
class Shape                             //定义形状类
{
public:
    double  area;                       //形状的面积
    void    virtual CaluateArea() = 0;  //计算面积
    void    virtual PrintArea() = 0;    //输出面积
};
```

抽象类的使用是有严格规定的。它既不能作为变量、成员数据和参数类型，也不能作为函数的返回值，并且不能显式地转换抽象类。在抽象类的构造函数中不能调用纯虚函数，包括间接地调用也不允许，但是可以调用其他成员函数。下面代码显示了抽象类的使用：

```
01  class base                          //声明基类
02  {
03  public:
04      base() {}                       //声明基类的构造函数
```

```
05      virtual ~base()=0;                    //声明基类的析构函数
06   };
07   base::~base() {}                         //定义基类的析构函数
08   class derived:public base                //声明派生类
09   {
10   public:
11       derived() {}                         //声明派生类的构造函数
12       ~derived(){}                         //声明派生类的析构函数
13   };
14   void main()                              //测试主函数
15   {
16       derived *pDerived = new derived;//定义派生类类型的变量
17       delete pDerived;                     //删除派生类
18   }
```

当删除 pDerived 时，调用派生类的析构函数后，调用基类的析构函数。纯虚函数的析构函数，保证有析构函数存在，即 base::~base()函数被隐含在 derived::~derived()函数中调用。在构造函数和析构函数外，使用完整的成员函数名可以显式地调用纯虚函数。

4.5　重载运算符

运算符是对象常用操作的简化。比较运算符是对象常用的运算符，用于比较两个对象是否相等。为了简化类的实现，C++提供了运算符重载机制，使类可以方便地根据实际情况，重新实现运算符的运算规则。本节将介绍运算符重载技术。

4.5.1　运算符重载语法

重载运算符通过函数来实现，可以是类成员函数，也可以是全局函数。重载运算符的名称为 operatorX，其中 X 是可以重载的运算符。如 operator+重载运算符是重载加法运算符，要重载此运算符，需要定义一个名为 operator+的函数。代码如下：

```
class Point                                     //声明表示点的类 Point
{
public:
    Point operator<( Point & );                 //声明操作符成员
    friend Point operator+( Point&, int );      //声明加法运算符
    friend Point operator+( int, Point& );      //声明减法运算符
};
```

上面代码中声明了作为成员函数的小于运算符和作为全局函数的加号运算符，其中加号运算符使用友元访问的方式实现。一个运算符可以有多个实现，即重载。如上面代码中的加号运算符有两个实现，不同之处是交换了参数的位置。虽然编译器在代码中遇到这些运算符时，会隐式地调用运算符重载函数，但是在代码中也可以像调用其他函数一样显式地调用这些运算符重载函数。代码如下：

```
Point pt;                        //定义 Point 类型的变量
pt.operator+( 3 );               //显式调用
```

当一元运算符重载时，声明为不带参数的成员函数。如果声明为全局函数，则需要带一个参数。当二元运算符重载时，声明为带一个参数的成员函数。如果声明为全局函数，则需要带两个参数。重载操作符不能使用默认参数。除了赋值运算符外，其他所有重载运算符都可以被派生类继承。运算符遵守优先权规则和分组规则，并且操作数的个数由使用内建类型的个数确定。虽然可以通过重载运算符改变运算符的含义，例如，将小于运算符修改成小于等于功能的实现，但是从编写程序的易维护性来讲，不建议这样做。

4.5.2　可重载的运算符

C++中可以重载大部分的内建运算符。这些运算符既可以全局重载，也可以基于类重载。表 4-2 中列出了可重载的运算符。

表 4-2　可重载的运算符

运　算　符	名　　称	类　　型
,	逗号	二元运算符
!	逻辑非 Logical NOT	一元运算符
!=	不等 Inequality	二元运算符
%	取模 Modulus	二元运算符
%=	取模赋值 Modulus/assignment	二元运算符
&	位与 Bitwise AND	二元运算符
&	地址符 Address-of	一元运算符
&&	逻辑与 Logical AND	二元运算符
&=	位与赋值 Bitwise AND/assignment	二元运算符
( )	函数调用	无
*	乘法	二元运算符
*	指针引用	一元运算符
*=	乘法赋值	二元运算符
+	加法	二元运算符
+	一元加法	一元运算符
++	自增	一元运算符
+=	加法赋值	二元运算符
–	减法	二元运算符
–	一元减	一元运算符
– –	自减	一元运算符
–=	减法赋值	二元运算符
–>	指针成员选择	二元运算符
–>*	指针成员选择指针符	二元运算符
/	除法	二元运算符
/=	除法赋值	二元运算符
<	小于	二元运算符
<<	左移	二元运算符
<<=	左移赋值	二元运算符
<=	小于等于	二元运算符

续表

<table>
<tr><th>运　算　符</th><th>名　　称</th><th>类　　型</th></tr>
<tr><td>=</td><td>赋值</td><td>二元运算符</td></tr>
<tr><td>==</td><td>判断是否相等</td><td>二元运算符</td></tr>
<tr><td>></td><td>大于</td><td>二元运算符</td></tr>
<tr><td>>=</td><td>大于等于</td><td>二元运算符</td></tr>
<tr><td>>></td><td>右移</td><td>二元运算符</td></tr>
<tr><td>>>=</td><td>右移赋值</td><td>二元运算符</td></tr>
<tr><td>[ ]</td><td>数组元素选择符</td><td>无</td></tr>
<tr><td>^</td><td>异或</td><td>二元运算符</td></tr>
<tr><td>^=</td><td>位异或赋值</td><td>二元运算符</td></tr>
<tr><td>|</td><td>位或</td><td>二元运算符</td></tr>
<tr><td>|=</td><td>位或赋值</td><td>二元运算符</td></tr>
<tr><td>||</td><td>逻辑或</td><td>二元运算符</td></tr>
<tr><td>~</td><td>按位补</td><td>二元运算符</td></tr>
<tr><td>delete</td><td>删除</td><td>无</td></tr>
<tr><td>new</td><td>新建</td><td>无</td></tr>
</table>

表 4-3 中列出了不可重载的运算符。

表 4-3　不可以重载的运算符

运　算　符	名　　称
.	对象成员选择符
.*	对象成员指针选择符
::	范围确定符
? :	条件运算符
#	预处理符号
##	预处理符号

4.5.3　重载赋值运算符

在重载运算符中，赋值运算符是比较特殊的。虽然赋值符也是二元运算符，声明的方法与其他的二元运算符是一样的，但是赋值运算符的使用与其他运算符之间有不同之处。赋值运算符必须是非静态成员函数，并且不能全局重载，而且派生类不能继承基类的赋值运算符。并且当没有重载赋值运算符时，编译器会生成默认的赋值运算符。以下代码显示了如何声明赋值运算符。

```
01  class Point                                 //声明代表点的 Point 类型
02  {
03  public:
04      Point & operator=( Point & );           //右边的是赋值参数
05  };
06  Point & Point::operator=( Point &pt )       //定义赋值运算符
07  {
08      _x = pt._x;                             //将传入的参数的 x 坐标赋值给变量
09      _y = pt._y;                             //将传入的参数的 y 坐标赋值给变量
```

```
10      return *this;                              //赋值运算符返回左边的对象
11  }
```

上面代码定义了 Point 类，并重载了赋值运算符，将赋值参数的_x 和_y 赋值给对象。

4.6　输入输出流库

输入输出是大部分程序需要用到的功能，为了提高开发效率，C++将有关输入输出的操作封装在库中，这样开发人员就可以轻松地实现输入输出功能。输入输出流库支持文本文件的处理、二进制磁盘文件和屏幕的输入输出功能。本节将介绍输入输出流库的使用。

4.6.1　C++的输入输出

C 和 C++都没有内建的输入/输出功能，但是 C++编译器会自动绑定系统面向对象的 I/O 包，即输入输出流库。核心概念是“流”，流对象可以看作实现从源到目的地的字节流。流的特性由类的插入符和提取符的重载实现。设备驱动、磁盘操作系统、键盘、屏幕、打印机和通信端口都可以看作扩展文件。使用输入输出流库可以与这些扩展文件进行交互，像操作磁盘文件一样从中读写数据。

C++输入输出流库中最主要的类是 iostream 类，它不是 C 运行时函数，而是面向对象的输入输出流类。可以实现缓冲的、带有格式化文本的输入输出流，也可以实现非缓冲的或二进制的输入输出流。同时还可以根据需要，从 iostream 类派生自定义的流类，实现特殊的输入输出功能。输入输出流库中还包括 cin、cout、cerr 和 clog 对象，分别表示输入对象、输出对象、错误对象和日志对象。如果将其与 QuickWin 库连接，则 cin、cout、cerr 和 clog 对象还可以与预定义的 stdin、stdout 和 stderr 相连，实现标准输入输出。如表 4-4 列出了输入输出流库中使用的类。

表 4-4　输入输出流库中的主要的类

类　名　称	类	说　　明
抽象流基类	ios	流基类
输入流类	istream	通用输入数据流类，是其他输入流的基类
	ifstream	输入文件数据流类
	istream_withassign	cin 对应的输入数据流类
	istrstream	字符串输入数据流类
输出流类	ostream	通用输出数据流类，是其他输出数据流的基类
	ofstream	输出文件数据流类
	ostream_withassign	cout、cerr 和 clog 对应的输出数据流类
	ostrstream	字符串输出数据流类
输入输出流类	iostream	通用输入/输出数据流类，是其他输入/输出数据流的基类
	fstream	输入/输出文件数据流类
	strstream	输入/输出字符串数据流类
	stdiostream	标准 I/O 对应的输入/输出数据流类
数据流缓冲区类	streambuf	数据流缓冲区抽象基类

续表

类 名 称	类	说　　明
数据流缓冲区类	filebuf	磁盘文件数据流缓冲区类
	strstreambuf	字符串数据流缓冲区类
	stdiobuf	标准 I/O 文件数据流缓冲区类
预定义流的初始化类	iostream_init	预定义流的初始化类

4.6.2　预定义输入/输出对象 cout 和 cin

输入输出流库中预定义了输出对象和输入对象，即 cout 和 cin。引入此文件，可以直接使用 cout 对象和 cin 对象。其中，cout 对象实现向标准输出对象输出内容的功能。cin 对象实现从标准输入设备提取数据的功能。其定义格式为：

```
extern ostream cout;
extern istream cin;
```

虽然 cout 和 cin 可理解为输出输入对象，但是实际上，cout 和 cin 是运算符重载函数，分别使用<<和>>输出输入数据。代码如下：

```
cin >> 存放输入信息的变量
cout<<["内容"|内容变量]
```

在上面代码中，第一条语句可以将标准输入设备输入的内容存放到变量中。第二条语句可以直接向标准输出设备输出内容字符串或存放在变量中的数据。

4.6.3　标准错误处理对象 cerr

输入输出流库中预定义了 cerr 对象，实现标准错误输出功能，标准错误输出对象可以是写入文件也可以是输出到屏幕。其定义格式为：

```
extern ostream cerr;
```

如下代码为向屏幕输出错误信息：

```
cerr << "您输入的年龄值不在有效范围内，请输入 0～120 之间的值";
```

4.6.4　常用输入输出成员函数

cin 对象提供了常用的输入输出成员函数，如表 4-5 所示。

表 4-5　常用输入输出流成员函数

	函 数 名	功　　能
输入成员函数	get()	从输入流获取字符，但是不包括分隔符
	getline()	从输入流获取一行字符，直到遇到分隔符为止
	read()	从输入流中获取数据

续表

	函　数　名	功　　能
输入成员函数	ignore()	忽略接收到的数据
	peek()	跳过接收到的字符，但是不忽略此数据
	gcount()	返回字符集合中的字符个数
输出成员函数	put()	向输出流插入单个字符
	write()	向输出流插入一系列字符
	flush()	缓冲输出流中的数据
	seekp()	移动数据流的指针位置
	tellp()	获取输出流当前指针的位置

使用输入输出流函数的方法与使用其他函数的方法是一样的。示例代码如下：

```
01  void ioTest()                      //使用输入输出流成员函数
02  {
03      char x;                        //定义存放输入字符的变量
04      while(cin.get(x))              //循环接收输入的字符
05      {
06          if (x == 'q') return;      //如果输入的是 q 字符，则退出程序
07          //如果输入的是大写字母，则在屏幕回显输入的字符
08          if ((x >'A') && (x <'Z')) cout.put(x);
09      }
10      system("pause");               //暂停程序，直到用户按下按键
11  }
```

上面代码循环接收屏幕输入的字符，并判断输入的字符是否为 q 字符，如果是，则退出程序。如果字符是大写字母，则在屏幕上回显用户输入的字符，否则，忽略用户输入的字符。

4.6.5　常见文件流类

fstream 类是派生自 iostream 类的实现磁盘文件输入输出功能的类；ifstream 类是派生自 iostream 类的实现磁盘文件输入功能的类；ofstream 类是派生自 iostream 类的实现磁盘文件输出功能的类。这 3 个类的构造函数会自动地创建和附加一个 filebuf 缓冲区对象。虽然 filebuf 对象的输入缓冲区和输出缓冲区在理论上是独立的，但是实际上输入缓冲区和输出缓冲区是不能同时激活的。当数据流从输入模式变为输出模式时，会清空输入缓冲区并重新初始化输出缓冲区。当数据流模式从输出模式变为输入模式时，会缓冲输出缓冲区并重新初始化输入缓冲区。文件流提供了有关文件的常用操作，如表 4-6 所示。

表 4-6　文件流中的常用函数

函　数　名	功　　能
open()	打开文件并将其附加到 filebuf 对象
close()	缓冲输出并关闭流对应的文件
setbuf()	将指定的预留区域附加到流的 filebuf 对象
setmode()	设置文件流的数据模式为二进制模式还是文本模式
attach()	通过 filebuf 对象将数据流附加到打开的文件
rdbuf()	获取数据流对应的 filebuf 对象

续表

函　数　名	功　　能
fd()	返回与数据流相连的文件描述
is_open()	判断数据流对应的文件是否打开

类 filebuf 派生自 streambuf 类，用于实现输入输出缓冲区的管理。filebuf()成员函数调用运行时库的底层输入输出函数，如_sopen()、_read()和_write()等函数。文件流类使用 filebuf 类的成员函数实现字符存取。下面两小节将具体介绍如何使用文件流读写文件。

4.6.6　操作顺序文件

顺序文件是将内容按照顺序存放在文件中，只需要存储数据内容即可，不需要存储其他辅助信息，因此，特点是占用存储空间少，但是由于没有任何索引信息，所以，在查找检索数据时，顺序文件的存储方式效率较低。适合存储不需要进行检索的数据，比如，记录程序运行日志。以下代码显示了使用文件流写顺序文件的过程。

```
01  void WriteOrderFile()                    //写顺序文件
02  {
03      ofstream myFile;                     //定义写文件流
04      myFile.open( "data.txt", ios::out);//打开文件
05      if (!myFile)                         //判断打开文件是否成功
06      {
07          cout << "打开文件错误" << endl;
08          return;
09      }
10      //向文件顺序写入两条数据
11      myFile << "01--张三" << endl << "02--李四" << endl;
12      myFile.close();                      //关闭文件
13      cout << "向 data.txt 文件写入两条数据." << endl;
14  }
```

上面代码首先定义了 ofstream 变量 myFile，然后调用 open()函数打开文件，并判断是否打开成功。如果打开文件成功，则使用<<操作符向文件中顺序写入数据，最后关闭文件，并向屏幕输出提示信息。使用文件流读取顺序文件的代码如下：

```
01  void ReadOrderFile()             //读顺序文件
02  {
03      ifstream myFile;             //定义读文件流
04      myFile.open( "data.txt", ios::in );//打开文件
05      if (!myFile)                 //判断打开文件是否成功
06      {
07          cout << "打开文件错误" << endl;
08          return;
09      }
10      cout << "读取 data.txt 文件内容如下所示:" << endl;
11      int value;
12      //使用 get()函数顺序从文件流中读取字符并显示
13      while((value = myFile.get()) != EOF)
14      {
15          cout << (char)value;
16      }
```

```
17        myFile.close();                //关闭文件
18    }
```

上面代码首先定义了 ifstream 变量 myFile，然后调用 open()函数打开文件，并判断是否打开成功。如果打开文件成功，则使用 get()函数依次从顺序文件中读出数据，最后关闭文件。

编写主函数并执行，代码如下，程序运行效果如图 4-8 所示。

```
01    #include <iostream>
02    #include <fstream>
03    using namespace std;
04
05    void WriteOrderFile()                        //写顺序文件
06    {
07        ...
08    }
09    void ReadOrderFile()                         //读顺序文件
10    {
11        ...
12    }
13
14    int main()
15    {
16        WriteOrderFile();
17        ReadOrderFile();
18        system("pause");
19        return 0;
20    }
```

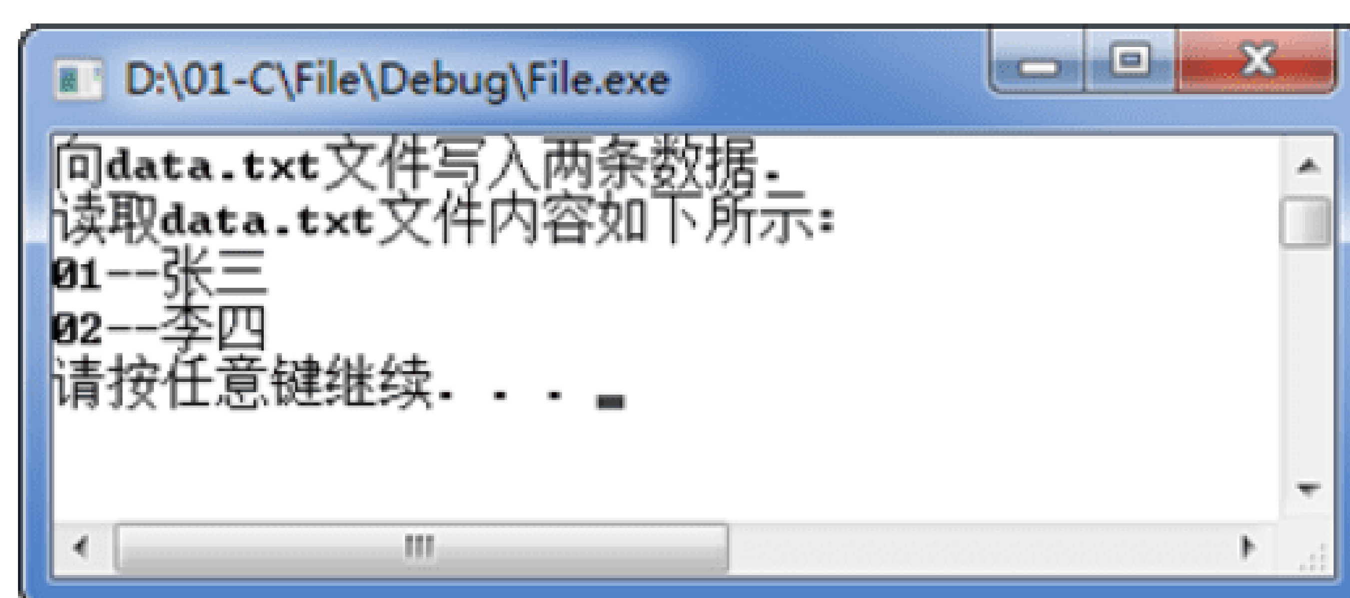


图 4-8　顺序文件操作运行效果

4.6.7　操作随机文件

随机文件是在数据内容前加上记录编号标识变长记录，或者使用固定长度的存储数据记录。读写效率比顺序文件要低些，而且占用的存储空间也较大，但是检索效率要比顺序文件高许多，适合用在经常需要检索的情况下。比如，要存取结构相同的记录时，使用随机文件比较合适。使用文件流写随机文件的代码如下：

```
01    struct Record                                //记录结构
02    {
03        char value[10];                          //存放数据内容的成员变量
04    };
05    void WriteRandomFile()                       //写随机文件
06    {
```

```
07      //定义要写入的数据
08      Record records[3] = {"01-张三", "02-李四", "03-王五"};
09      ofstream myFile;                             //定义写文件流
10      //打开文件
11      myFile.open( "data.txt", ios::out | ios::binary);
12      if (!myFile)                                 //判断打开文件是否成功
13      {
14          cout << "打开文件错误" << endl;
15          return;
16      }
17      for (int i = 0;i < 3;i++)                    //依次写入 3 条记录
18      {
19          //调用 write()函数写入数据
20          myFile.write((const char*)&records[i], sizeof(Record));
21      }
22      myFile.close();                              //关闭文件
23      cout << "向 data.txt 文件中写入三条数据." << endl;
24  }
```

上面代码首先定义了要写入的数据变量和 ofstream 变量 myFile，然后调用 open()函数打开文件，并判断是否打开成功。如果打开文件成功，则使用 for 循环调用 write()函数依次写入 3 条记录，最后关闭文件，并向屏幕输出提示信息。使用文件流读取随机文件的代码如下：

```
01  void ReadRandomFile()                  //读随机文件
02  {
03      ifstream myFile;                   //定义读文件流
04      myFile.open( "data.txt", ios::in | ios::binary );
05      if (!myFile)                       //判断打开文件是否成功
06      {
07          cout << "打开文件错误" << endl;
08          return;
09      }
10      cout << "读取 data.txt 文件的第二条内容如下所示:" << endl;
11      Record record;                     //定义存放获取的记录的变量
12      //把文件的写指针从文件开头向后移一条记录
13      myFile.seekg(sizeof(Record), ios::beg);
14      //读取文件中的第二条记录
15      myFile.read((char*)&record, sizeof(record));
16      cout  << (char*)&record;           //将获取的记录显示在屏幕上
17      cout  << endl;
18      myFile.close();                    //关闭文件
19  }
```

上面代码首先定义了 ifstream 变量 myFile，然后调用 open()函数打开文件，并判断是否打开成功。如果打开文件成功，则先使用 seekg()函数把文件的写指针从文件开头向后移一条记录，并调用 read()函数读取文件中的第二条记录，输出到屏幕上，最后关闭文件。

编写主函数并执行，代码如下，程序运行效果如图 4-9 所示。

```
01  #include <iostream>
02  #include <fstream>
03  using namespace std;
04
05  struct Record                                //记录结构
```

```
06  {
07      char value[10];                              //存放数据内容的成员变量
08  };
09  void WriteRandomFile()                           //写随机文件
10  {
11      ...
12  }
13  void ReadRandomFile()                            //读随机文件
14  {
15      ...
16  }
17  int main()
18  {
19      WriteRandomFile();
20      ReadRandomFile();
21      system("pause");
22      return 0;
23  }
```

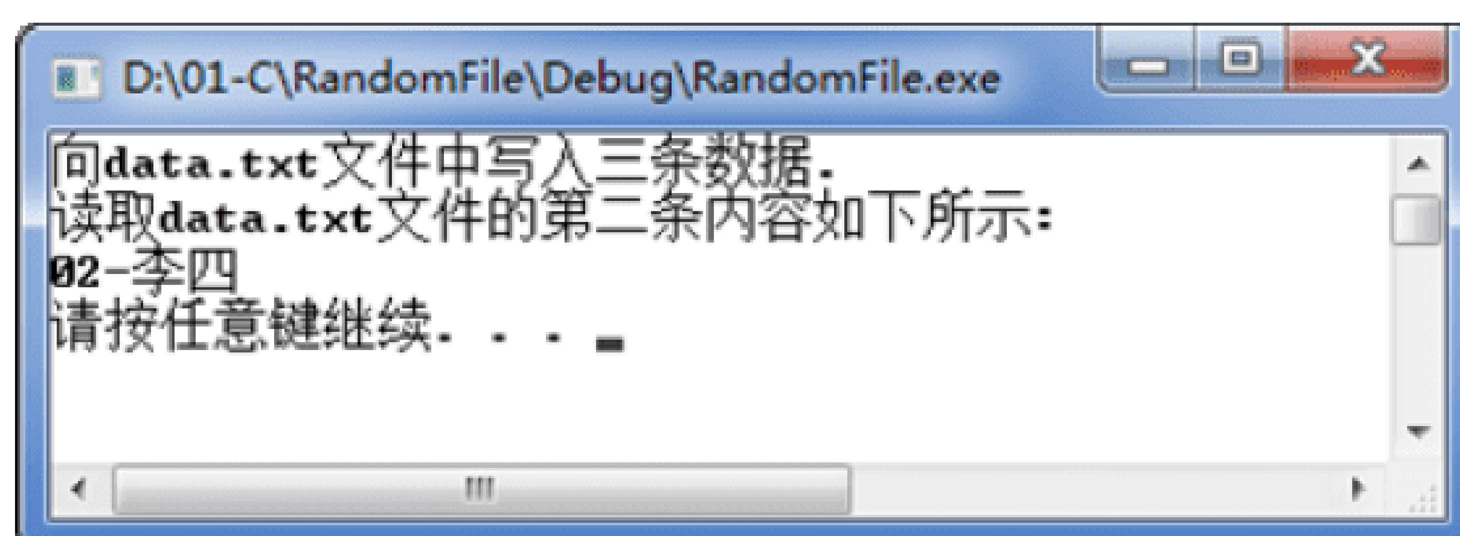


图 4-9　随机文件操作运行效果

4.7　C++的模板机制

为了实现参数化类和补充函数重载机制的不足，C++提供了模板机制。通过模板，可以编写实现多种类型相同功能的单个类，简化了开发工作量。在 VC 中，MFC 中的许多库都是使用模板机制实现的。本节将介绍模板的基础知识。

4.7.1　为什么需要模板

前面介绍过函数重载机制，可以实现相同函数的不同版本，从而简化相同功能间的函数复杂度。代码如下：

```
//min()函数的 int 版本
int min( int a, int b )        return ( a < b ) ? a : b;
//min()函数的 long 版本
long min( long a, long b )     return ( a < b ) ? a : b;
//min()函数的 char 版本
char min( char a, char b )     return ( a < b ) ? a : b;
```

上面的代码实现了min()函数，返回输入的两个参数中相对较小的那个数的值。这 3 个函数为 min()函数的不同重载版本。当增加新的数据类型支持时，需要增加相同功能的重载函数。为了解决这个问题，C++提供了模板机制，模板是基于类型参数的函数或类，也称

为“参数化类型”。使用模板，用户可以设计单个类或函数来操作多种类型的数据，而不必为每种类型创建一个单独的类或函数，从而大大减少了代码量，并提高了代码的灵活性。以下代码实现了上面 3 个函数的函数模板：

```
//min()函数的模板版本
template <class T>
T min( T a, T b )  return ( a < b ) ? a : b;
```

使用模板不需要手动处理每种数据类型，因此易于编写。由于抽象了对象行为，因此更易于理解。同时，因为模板使用的类型在编译时就知道，因此又是类型安全的。所以，使用模板优化代码是一种重要的手段。

4.7.2　函数模板的使用

函数模板是指定义的一组可以操作不同类型信息的函数的模板。如用户使用函数模板创建一组函数应用于不同数据类型的相同的数据运算。有些情况下，模板比宏和指针使用起来更方便。例如，MFC 中的集合类都是通过模板实现的。函数模板的语法格式如下：

```
template < [类型列表] [, [ 参数列表 ]] > 声明
```

其中，template 关键字指定定义的函数为模板函数。参数列表是以逗号分隔的类型列表。声明是函数声明。以下代码是交换两个变量的模板函数的定义。

```
template <class T>
void MySwap( T& a, T& b )                  //交换取值的函数模板
{
   T c( a );                               //定义 T 类型的中间变量
   a = b; b = c;                           //交换传入的两个参数值
}
```

上面代码定义了一组函数，用于交换两个参数的值。通过这个模板，不仅可以生成用于交换 int 和 long 类型的数据，还可以用于生成交换用户自定义数据类型的数据。只要正确定义了类的复制函数和赋值函数，就可以使用 MySwap()函数交换两个类对象。而且，上面的模板也隐含着要交换的两个对象必须是相同类型的。调用模板函数与调用普通函数的方法是相同的，不需要特殊的语法，如以下代码所示：

```
int i, j;
char k;
MySwap( i, j );      //调用正确
MySwap( i, k );      //错误，类型不同
```

上面代码显示了如何调用模板函数。从中也可以看出，调用模板类型时，要注意参数类型的一致性。当然也允许显式地指定函数模板的模板参数类型，如以下代码所示：

```
template<class T>
void f(T) {...}                            //此处代码省略
void g(char j)
{
  f<int>(j);                               //调用指定类型的模板
}
template<class T>
```

```
void m(T, int){ };                              //带部分参数的模板
int i;
char c;                                         //定义模板参数
m(i, c);                                        //调用模板
```

上面代码显式地指定 f()模板函数使用 int 类型，即编译器会将 char 类型的 j 转换成 int 类型。而变量 i 转换成类型 T 时可能会发生错误，但是允许 c 转换成 int 类型。

每当第一次调用函数模板的一个类型时，编译器会创建此类型对应的“实例”，是一个指定类型的模板函数版本。每次使用此类型的函数模板时，都会调用此实例。如果同时有几个相同的实例，不管是否在相同的模板中，只存放一个实例副本。因此，即使显式地指定函数模板对效率也不会有影响，如以下语句不会影响运行效率。

```
template<class T>
void f(T) {...}                                 //模板 f，此处代码省略
template void f<int> (int);                     //显式指定模板参数为 int 类型
template void f(char);                          //隐式指定模板参数为 char 类型
```

4.7.3　类模板的使用

使用类模板可以实现一组类型安全的类。以下代码显示了类模板的定义方法。

```
01  template <class T, int i>
02  class TempClass                                     //定义模板类 TempClass
03  {
04  public:
05      TempClass( void );                              //声明模板类的构造函数
06      ~TempClass( void );                             //声明模板类的析构函数
07      int MemberSet( T a, int b );                    //带模板参数的成员函数
08  private:
09      T Tarray[i];                                    //模板数组
10      int arraysize;                                  //数组大小
11  };
```

在上例中，模板类使用两个参数，一个是类型 T 和一个整型 i。T 参数可以传入任何类型，包括结构和类。i 参数被作为整型常数传入。因为 i 是在编译时定义的常数，用户可以使用标准数组声明定义成员数组的大小 i。

模板类的成员函数与非模板类的成员函数定义是不同的，代码如下：

```
01  template <class T, int i>
02  TempClass< T, i >::TempClass( void )            //构造函数
03  {
04      TRACE( "创建 TempClass.\n" );               //打印提示信息
05  }
06  template <class T, int i>
07  TempClass< T, i >::~TempClass( void )           //析构函数
08  {
09      TRACE( "释放 TempClass.\n" );               //打印提示信息
10  }
11  //定义带模板参数的成员函数
12  template <class T, int i>
13  int TempClass< T, i >::MemberSet( T a, int b )
14  {
```

```
15      if( ( b >= 0 ) && (b < i) )                //判断输入的数组索引值是否有效
16      {
17          Tarray[b++] = a;                       //将输入的数组元素加入到数组中
18          return sizeof( a );                    //返回当前数组的大小
19      }
20      else
21          return -1;                             //否则返回-1，表示操作失败
22  }
```

上面代码定义了模板类的构造函数、析构函数和 MemberSet()成员设置函数的实现。实例化类模板的语法与实例化普通类是相同的，但是需要在尖括号内包括模板参数，并且在实例化时显式地指定类模板参数类型。以下代码定义了模板类 TempClass 的实例。

```
TempClass <char, 5> ClassInst;          //定义 TempClass 类的实例 ClassInst
TempClass<float, 6 > test1;             //定义 TempClass 类的实例 test1
TempClass<char, items++ > test2;        //调用错误，第二个参数必须为常数
```

类模板在第一次使用时才会由编译器进行实例化，只有实例化后，才会生成代码，成员函数只有在被调用时才会被实例化。

4.7.4　模板与宏的对比

模板与宏有很多相似之处，都是使用给定的类型替换模板化的变量。模板和宏也存在很多不同之处。例如以下代码：

```
#define min(i, j) (((i) < (j)) ? (i) : (j))
template<class T>
T min (T i, T j) { return ((i < j) ? i : j) }
```

在上面代码中，第一条语句定义了 min 宏，用于返回较小的数。第二条语句定义了模板 min，用于返回同类型中较小的数。这两者之间的区别在于：

- 第一条语句定义的宏，在预处理器编译宏时，不会进行类型检查，因此这时类型是不安全的。而第二条语句中的模板在编译时会进行类型安全检查。
- 如果任何一个参数有增量变量，则自增会执行两次操作。
- 因为宏是由预处理器展开，编译错误消息会指向展开的宏，而不是宏定义本身。因此在调试时，宏也会暴露出来，从而有可能造成代码泄露。

模板是实现集合类的好方法。MFC 类库中的 CArray、CMap、CList、CTypedPtrArray、CTypedPtrList 和 CTypedPtrMap 都是使用模板实现的。示例代码如下：

```
01  template <class T, int i>
02  class MyStack        //堆栈模板类
03  {
04      T StackBuffer[i];                               //堆栈缓冲区
05      int cItems;                                     //元素个数
06  public:
07      void MyStack( void ) : cItems( i ) {};          //声明获取堆栈元素的函数
08      void push( const T item );                      //声明元素入栈函数
09      T pop( void );                                  //声明元素出栈函数
10  };
11  //定义元素入栈函数
12  template <class T, int i>
```

```
13  void MyStack< T, i >::push( const T item )
14  {
15      //如果元素索引有效，则存入元素值
16      if( cItems > 0 )
17          StackBuffer[--cItems] = item;
18      else
19          throw "堆栈溢出错误.";                      //否则抛出异常
20      return;
21  }
22  //定义元素出栈函数
23  template <class T, int i>
24  T MyStack< T, i >::pop( void )
25  {
26      //如果元素索引有效，则弹出元素值
27      if( cItems < i )
28          return StackBuffer[cItems++]
29      else
30          throw "堆栈溢出错误.";                      //否则抛出异常
31  }
```

上面代码的 MyStack 集合实现了简单的堆栈功能。两个模板参数 T 和 i，分别指定堆栈中的元素类型和堆栈中的最大数量。push()成员函数和 pop()成员函数用来从堆栈中增加和移除数据项。

4.7.5　模板应用示例

C++中有一种“智能指针”类，封装指针并重载指针运算符为指针操作增加新功能。使用模板可以封装任何类型的指针。下面的代码演示了一个简单的“垃圾回收计数器”引用的实现。

```
01  class RefCount                                       //计数类
02  {
03      int crefs;                                       //定义计数变量
04  public:
05      //计数类的构造函数，初始化计数变量为 0
06      RefCount(void)
07      { crefs = 0; }
08      //析构函数，打印提示信息和当前计数值
09      ~RefCount()
10      { TRACE("退出 (%d)\n", crefs); }
11      //增加计数值
12      void upcount(void)
13      { ++crefs; TRACE("增加当前计数值=%d\n", crefs);}
14      void downcount(void)                             //减少计数值
15      {
16          if (--crefs == 0)
17              delete this;//如果计数值为 0，则删除对象
18          else
19              TRACE("减少当前计数值=%d\n", crefs);  //打印提示信息
20      }
21  };
22  class Sample : public RefCount
23  {                                                    //声明派生类
24  public:
```

```
25      void doSomething(void)
26      { TRACE("测试函数\n");}                              //定义测试函数
27  };
28  template <class T>
29  class Ptr                                                //定义模板类
30  {
31      T* p;                                                //定义模板参数对应的变量
32  public:
33      Ptr(T* p_) : p(p_) { p->upcount(); }                 //增加链表元素个数
34      ~Ptr(void) { p->downcount(); }                       //模板类析构函数
35      operator T*(void) { return p; }              //获取元素
36      T& operator*(void) { return *p; }                    //获取元素地址
37      T* operator|(void) { return p; }                     //获取元素指针
38      //重载等于操作符
39      Ptr& operator=(Ptr<T> &p_) {return operator=((T *) p_);}
40      //重载等于操作符
41      Ptr& operator=(T* p_)
42      {p|downcount(); p = p_; p->upcount(); return *this; }
43  };
44  int main() {
45      Ptr<Sample> p  = new Sample;        //创建参数类型为 Sample 的变量
46      Ptr<Sample> p2 = new Sample;        //创建参数类型为 Sample 的变量
47      p = p2;     //p 中的 crefs 值为 0，因此会销毁此对象。而 p2 的 crefs 的值为 2
48      p->doSomething();                   //调用 p 的 doSomething 测试函数
49      return 0;
50  }
```

类 RefCount 和 Ptr<T>一起提供了简单的垃圾回收解决方案。模板类 Ptr<T>实现从 RefCount 类继承而来的所有类型的垃圾回收指针。例如假定类用于创建和管理垃圾回收的文件、符号和字符串等内容，使用类模板 Ptr<T>，编译器会创建模板类 Ptr<File>、Ptr<Symbol>、Ptr<String>等和 Ptr<File>::~Ptr()、Ptr<File>::operator File*()、Ptr<String>::~Ptr()、Ptr<String>::operator String*()成员函数等。

4.7.6　C++标准模板库 STL 简介

STL（Standard Template Library，标准模板库）是 C++提供的一组常用的模板库。表 4-7 中列出了其中常用的模板。

表 4-7　STL中常用的模板

模 板 名 称	模 板 功 能
<algorithm>	定义多种模板实现常用的运算法则
<deque>	定义了实现队列容器的模板类
<functional>	定义了在<algorithm>和<numeric>中使用的一些基本模板类
<iterator>	实现定义和操作迭代器的模板类
<list>	实现列表容器的模板类
<map>	实现联合容器的模板类
<memory>	实现内存管理的模板类
<numeric>	实现数字函数的模板类
<queue>	实现队列容器的模板类
<set>	实现带有索引的联合容器的模板类

续表

模 板 名 称	模 板 功 能
<stack>	实现堆栈容器的模板类
<utility>	实现常用功能的模板类
<vector>	实现矢量容器的模板类

4.8　C++实例——设计一个电子时钟

前面几节介绍了 C++语法，本节以电子时钟实例来介绍如何使用 C++语言编写程序。程序的功能是根据用户输入的时间和计算机的时钟频率刷新当前的时间。代码如下：

```
01  #include <iostream>
02  using namespace std;
03
04  void ShowClock()                                    //显示电子时钟
05  {
06      int second = -1,minute = -1, hour = -1, delay;
07      cout << "请输入 24 小时制的起始时间(时:分:秒):" ;
08      cin >> hour >> minute >> second;                //获取用户输入的当前时间
09
10      while(true)
11      {
12          for (;second<=60;second++)
13          {
14              for (delay=0;delay<=220000000;delay++)
15              {
16                  continue;       //延时时间，此处根据 CPU 的始终频率，
17                                  //设置延时时间
18              }
19              if (second==60)     //如果秒为 60，则分加 1，秒重新变成 0
20              {
21                  minute++;
22                  second=0;
23              }
24              if (minute==60)     //如果分为 60，则小时加 1，分重新变成 0
25              {
26                  hour++;
27                  minute=0;
28              }
29              if (hour==24)       //如果小时为 24，则小时重新归为 0
30                  hour=0;
31              //在屏幕上输出时间
32              cout << "现在时间: "<< hour << ":"
33                     << minute << ":" <<  second << "\r" ;
34          }
35      }
36      system("pause");                    //暂停显示
37  }
38
39  int main()
40  {
41      ShowClock();
```

```
42      return 0;
43  }
```

上面的代码每延时一次 CPU 时钟频率时间间隔后，时钟秒数增加 1 秒。如果秒数达到 60，则分钟数增加 1；如果分钟数达到 60，则小时数增加 1；如果小时数达到 24，则清零重新计数。程序运行效果如图 4-10 所示。

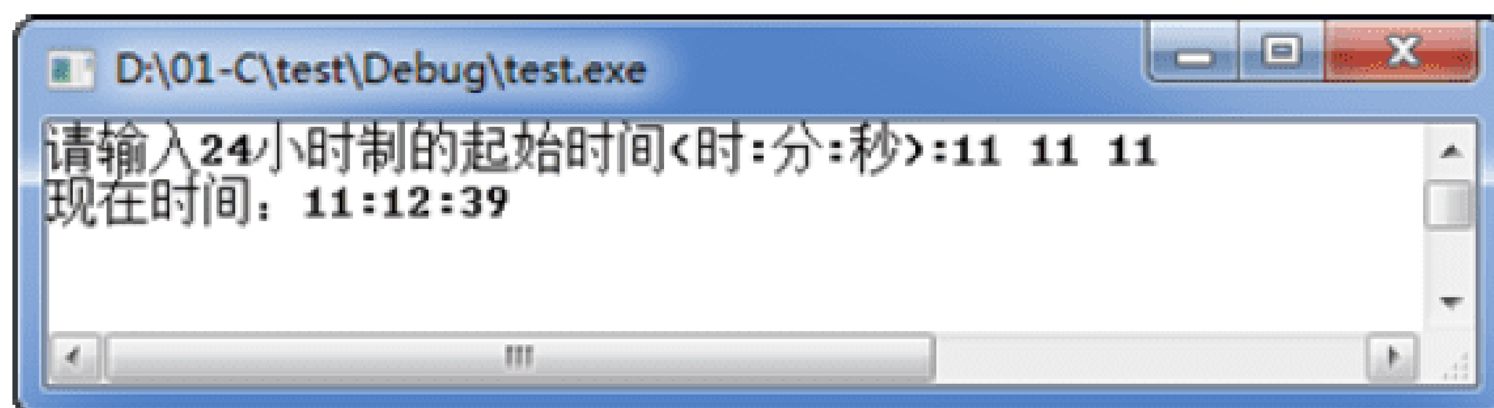


图 4-10 电子时钟运行效果

运行上面的示例，在界面上输入当前时间，则程序会每隔一定时间刷新当前时间，此处的当前时间不是计算机系统时间，而是根据 CPU 时钟计算出来的。因此，此程序可以作为定时器和电子时钟使用。

4.9 本章小结

本章介绍了 C++语言中非常重要的概念——类，并在此基础上介绍了类的成员及其特性。C++通过继承和虚函数提供了对面向对象程序设计思想的支持。运算符重载是 C++中常用的技术。本章还介绍了用于实现输入输出的输入输出流库。最后介绍了可以实现参数化类型的模板。本章重点是理解 C++面向对象思想，掌握 C++类的编写方法及输入输出流库的使用。本章难点是深刻理解模板的实现机制。第 5 章将讲解 Windows 编程与 MFC 基础的界面开发。

4.10 习题

1．有一个类 ClassAdd，它的定义如下：

```
class ClassAdd
{
private:
    int x;
    int y;
public:
    ClassAdd();
    void printMember();
};
ClassAdd::ClassAdd()
{
    x = y = 1;
}
void ClassAdd::printMember()
{
    cout << "x=" << x << " y=" << y << endl;
}
```

尝试使用外部函数 ModifyMember(ClassAdd &Ca，int a,int b)来修改类 ClassAdd 定义的对象 Ca 的私有数据成员 x 和 y。程序的运行效果可以是图 4-11 所示的样子。

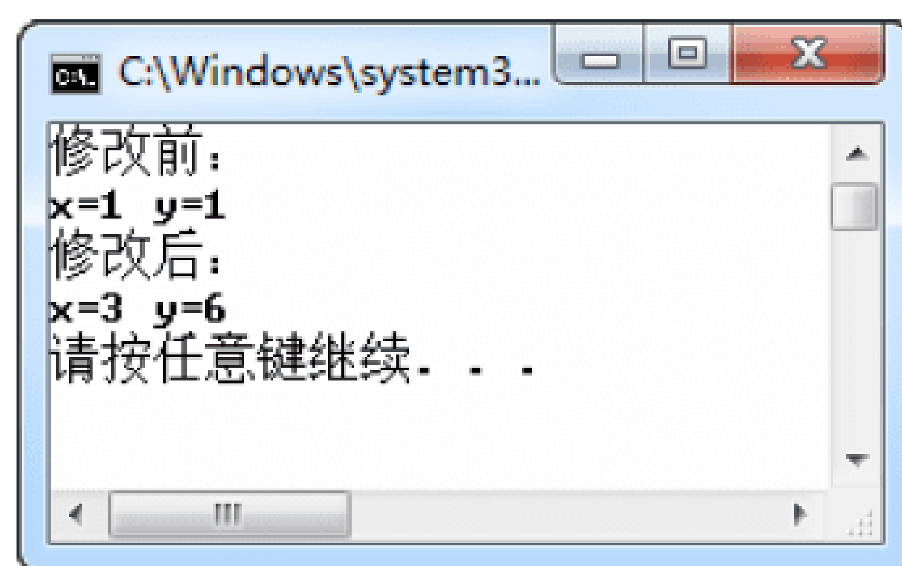


图 4-11　通过友元函数修改类的私有数据成员

【思路】通常情况下类的私有数据成员是不能被外部访问到的，但是也有例外——友元。可以为类 ClassAdd 添加友元函数。

2．已经定义了两个类 A 和 B，定义如下：

```
class A
{
protected:
    int a;
public:
    A(int x);
};
A::A(int x)
{
    a = x;
}
class B
{
protected:
    int b;
public:
    B(int y);
};
B::B(int y)
{
    b = y;
}
```

尝试定义一个类 C，它同时继承了类 A 和 B，它定义有自己的 int 型数据成员 c，它的一个公有的成员函数 printMember()用来输出它的所有的数据成员的值，即 a、b 和 c 的值。程序的运行效果可以是图 4-12 所示的样子。

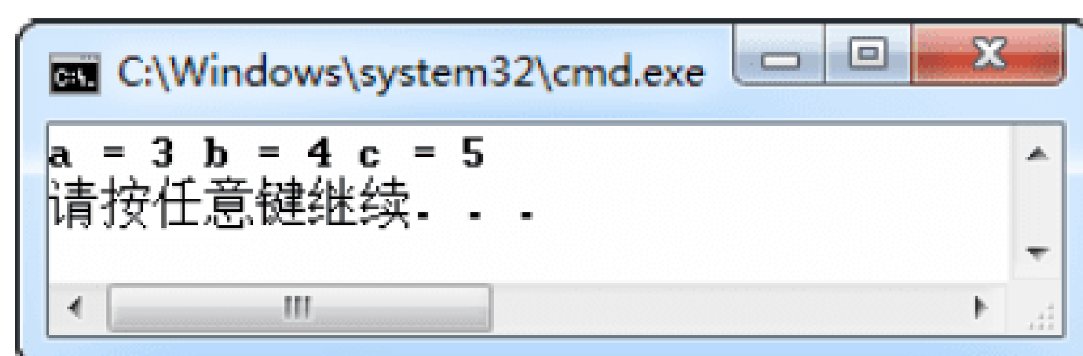


图 4-12　多重继承举例

【思路】类 A 和 B 的数据成员都是保护型的，所以可以在派生类中访问，那么可以这样设计类 C：在构造函数中完成对所有数据成员的赋值，在成员函数 printMember()中完成数据成员的输出。

第 2 篇　界面开发

第 5 章　Windows 编程与 MFC 基础

Windows 操作系统通过 Windows API 为开发人员提供访问操作系统底层功能的支持，如使用 Windows API 可以处理输入输出、编写驱动程序等。为了减轻重复开发的工作量，提高代码复用性，VC 提供了一组类库——微软基础类库 MFC（Microsoft Foundation Class），它将常用的功能封装成类。MFC 涵盖了程序功能的各个方面，包括支持基础类库的架构类、与对话框相关的类、有关通信编程类以及基本数据类型类等。本章首先介绍有关 Windows 编程的基本知识和 MFC 基础。

5.1　Windows 编程

Windows 操作系统中，显示给用户的界面接口是窗体，并使用句柄标识不同的窗体，通过事件和消息传递命令，并且使用消息队列按照消息发送的先后顺序处理。用户可以使用 Windows API 函数调用操作系统底层的功能，并且具有自有的常用数据类型。本节将介绍这些 Windows 编程的基础知识。

5.1.1　Windows 应用程序编程接口 API

Windows API 指 Windows 操作系统应用程序编程接口（Application Programming Interface，API）。它支持操作系统中的函数定义、参数定义、结构定义、消息格式、宏和接口等的实现，为微软 Windows 平台提供了统一的接口。因为各个 Windows 平台是有差异的，所以微软提供了多种版本的 Windows API。从较早的 Win16 API 到现在普遍使用的 Win32 API，在此期间不同平台下 API 函数的使用略有不同。本书以 Win32 API 编程接口为例进行介绍。

Win32 API 函数在各个平台上的主要不同之处在于，函数运行平台的限制和其他系统限制。如有关安全方面的函数只能在 Windows NT 操作系统上使用，而且函数可以使用的参数不同。

Win32 API 函数涵盖的范围很广，熟练使用这些函数可以完成各种功能，主要包括以下几个方面。

- Window 管理：完成 Windows 管理中的各方面功能。在第 20 章中会介绍有关 Window 管理中的部分函数的使用。
- Window 控件：完成标准 Windows 控件的功能。在第 7 章中会介绍有关 Windows 控件的使用。
- 系统内核：完成 Windows 操作系统的一些核心操作。在第 5 篇中会介绍有关系统

内核的 Window API 函数的使用。

- GDI 指图形设备接口：完成 Windows 操作系统中有关图形绘制的功能。在第 6 篇中会涉及有关 GDI 编程。
- 系统服务：提供对 Windows 操作系统底层服务的支持。
- 国际化支持：提供对多语言的支持。
- 网络服务：在第 4 篇中会讲解网络方面的编程知识。

5.1.2 使用句柄标识窗口

在图形化 Windows 应用程序中，窗口是应用程序显示在输出屏幕上的一个矩形区域，可以用于接收用户的输入，也可以显示程序的数据处理结果。因此，图形化 Windows 应用程序的第一步工作就是创建窗口。多个应用程序窗口可以共享同一屏幕。但是同一时间只有一个窗口可以通过鼠标、键盘或其他输入设备接收用户数据的输入，并由窗口所属的应用程序处理。

窗口有很多种形式，从对话框到编辑框再到程序的运行主界面都是窗口。Window API 中使用 HWND（窗口句柄类型）标识窗口。HWND 数据类型在 32 位操作系统中存储为一个 32 位的无符号整型值。要注意的是，因为各个平台下 HWND 的存储空间大小是不同的，所以计算句柄大小时，应该使用 sizeof（HWND）函数计算。

5.1.3 输入事件产生的消息

Windows 应用程序是事件驱动的，一般情况下不会显式地调用函数获取输入，而是等待系统将接收到的输入传递给应用程序。系统会将应用程序的输入传递给不同的窗口。每个窗口有一个对应的函数，称为窗口函数，当有对应窗口的输入时，系统会调用此函数。窗口函数会处理输入，并执行对系统的控制。

系统使用消息的形式将输入传递给窗口函数。系统和应用程序都可以创建消息。每发生一个输入事件，系统会产生一条消息。如当用户输入时，移动鼠标或单击控件都会产生输入事件。系统也会产生响应消息，用于响应应用程序发送给系统的消息，如当应用程序改变系统字体资源池时，或是重新调整窗口大小时，系统都会产生响应消息。应用程序可以产生指向其所属窗口的消息完成任务，也可以与其他应用程序的窗口通过消息进行通信。系统向窗口函数发送消息时，需要 4 个参数，如下所述。

- 窗口句柄：用于表示向哪个窗口发送消息，操作系统通过这个参数判断哪个窗口函数接收这条消息。
- 消息标识：是一个常数，用于表示消息的种类。当消息对应的窗口函数接收到一条消息时，会使用消息标识确定如何处理消息。如消息 WM_PAINT 表示窗口的工作区域内容发生变化了，需要重新绘制，而消息 WM_TIMER 表示，对应窗口的定时器事件发生了，由应用程序确定如何处理定时器事件。
- 两个消息参数：在 32 位操作系统下，都是 32 位的值。用于表示消息所附带的参数，其含义和取值根据消息的不同有所不同。而消息参数可以是整型，也可以是指向包含更多数据的结构的指针等。当不使用消息参数时，一般将其设置为 NULL。窗口函数通过检查消息标识来决定如何解释消息参数。

Windows 操作系统使用以下两种方式将消息传递给窗口函数。

- 发送消息到先进先出的消息队列中，用于存储通过鼠标或键盘输入的用户输入，如 WM_MOUSEMOVE、WM_LBUTTONDOWN、WM_KEYDOWN 和 WM_CHAR 等消息；也用于存储定时器消息（WM_TIMER）、重绘消息（WM_PAINT）和退出消息（WM_QUIT）等。通过消息队列发送的消息称为队列消息。使用 PostMessage()函数发送的消息会发送到消息队列中。
- 使用系统定义的内存对象临时存储消息，并将消息直接发送给窗口函数。除了上面的这些队列消息外，一般情况下，其他消息都采用这种方式处理。使用 SendMessage()函数发送的消息会直接发送给窗口函数。

5.1.4　Windows 句柄的数据类型

在 Windows API 中，使用 Windows 数据类型定义函数的返回值类型、参数类型和消息参数以及结构成员的类型。它定义了这些元素的大小和含义，主要分为字符型、整型、布尔型、指针和句柄 5 种类型。字符性、整型和布尔型是 C 编译器常用的类型。大部分指针类型的数据类型都以 P 或 LP 为前缀。句柄用于代表内存中的资源。表 5-1 列出了常用的 Windows 句柄数据类型。

表 5-1　常用的Windows句柄数据类型

类　型	定　义	类　型	定　义
HACCEL	加速键表句柄	HHOOK	钩子句柄
HANDLE	对象句柄	HICON	图标句柄
HBITMAP	位图句柄	HIMAGELIST	图像列表句柄
HBRUSH	画刷句柄	HINSTANCE	实例句柄
HCURSOR	光标句柄	HKEY	注册表项句柄
HDC	设备上下文句柄	HKL	键盘布局句柄
HDDEDATA	DDE 数据句柄	HLOCAL	本地内存块句柄
HDESK	桌面句柄	HMENU	菜单句柄
HDROP	内部下拉结构句柄	HMETAFILE	元文件句柄
HDWP	窗口位置结构句柄	HMODULE	模块句柄
HENHMETAFILE	增强型图元句柄	HMONITOR	显示器句柄
HFILE	文件句柄	HPEN	铅笔句柄
HFONT	字体句柄	HRGN	区域句柄
HGDIOBJ	GDI 对象句柄	HRSRC	资源句柄
HGLOBAL	全局内存块句柄	HWND	窗口句柄

表 5-1 中列出了 Windows API 中使用的各种资源的句柄类型，如 HWND 表示窗口句柄，HMENU 表示菜单句柄，这些资源可以标识 Windows 操作系统中对应的各种资源。

5.2　Windows 程序执行流程

本节将介绍 Windows 应用程序的执行流程。核心要点包括程序入口函数、窗口菜单、

窗口函数以及关于对话框等内容。最后以一个基于 Win32 的应用程序为例，详细说明 Windows 应用程序的实现。通过这个实例，可以初步了解 Windows 应用程序的结构。

5.2.1　入口函数 WinMain()

每个 Windows 应用程序都必须具有一个程序开始执行点——入口函数。默认情况下，入口函数的名称为 WinMain。在 Win32 平台下的函数声明为：

```
int APIENTRY WinMain(
    HINSTANCE   hInstance,       //指定应用程序的当前实例句柄
    HINSTANCE   hPrevInstance,   //指定应用程序前一个实例的句柄，默认为 NULL
    LPSTR   lpCmdLine,           //以非 NULL 结束的字符串用于指定执行程序的
                                 //应用程序命令行
    int  nCmdShow)               //应用程序主对话框如何显示
```

其中，nCmdShow 参数用于指定窗体的显示方式，有效取值如表 5-2 所示。

表 5-2　对话框显示方式

取　　值	显 示 方 式
SW_HIDE	隐藏对话框并激活其他对话框
SW_MINIMIZE	最小化指定对话框，并激活系统列表中最顶层的对话框
SW_RESTORE	激活并显示对话框。如果对话框是在最小化或最大化状态，则系统恢复原始大小和位置，与 SW_SHOWNORMAL 参数的作用一样
SW_SHOW	激活对话框，并显示当前大小和位置
SW_SHOWMAXIMIZED	激活对话框，并将对话框最大化显示
SW_SHOWMINIMIZED	激活对话框，并将对话框以图标形式显示
SW_SHOWMINNOACTIVE	显示对话框图标，并且激活的对话框仍然保持激活状态
SW_SHOWNA	以当前的状态显示对话框，激活的对话框仍然保持激活状态
SW_SHOWNOACTIVATE	以最近的大小和位置显示对话框，激活的对话框仍然保持激活状态
SW_SHOWNORMAL	激活并显示对话框。如果对话框是最小化状态或最大化状态，则系统恢复原始大小和位置，与 SW_RESTORE 作用相同

WinMain()函数会初始化应用程序，显示程序的主对话框、进入消息接收和调度循环，直到收到 WM_QUIT 消息。当收到 WM_QUIT 消息后程序会终止，并且会将消息传入的 wParam 参数包含的退出代码值返回。如果在进入消息循环之前终止，则会返回 0。WinMain()函数主要完成 3 个工作：注册窗体类、创建窗口和启动消息循环。下面 3 个小节分别介绍这 3 个工作。

5.2.2　注册窗体类

每个窗体必须有一个与其对应的窗体类。窗体类定义了窗体的属性，如窗体样式、图标、光标、菜单名和窗体函数名称等。因此，在入口函数中的第一步就是注册程序的主窗体类。注册窗体类的过程分两步：首先使用类信息初始化 WNDCLASS 对象，指定窗体的属性；然后将结构传入 RegisterClassEx()函数，在系统中注册对应的窗体类。函数 RegisterClassEx()封装如下：

```
01  ATOM MyRegisterClass(HINSTANCE hInstance)      //注册窗体类的函数
02  {
03      WNDCLASSEX  wcex;                          //定义注册的窗体类的结构变量
04      wcex.cbSize          = sizeof(WNDCLASSEX);  //为结构大小成员赋值
05      wcex.style           = CS_HREDRAW | CS_VREDRAW;//赋值窗体的样式成员
06      wcex.lpfnWndProc     = (WNDPROC)WndProc;    //赋值窗体的处理函数
07      wcex.cbClsExtra      = 0;                   //赋值窗体类的数据长度
08      wcex.cbWndExtra      = 0;                   //赋值窗体的数据长度
09      wcex.hInstance       = hInstance;           //赋值窗体类的实例句柄
10      wcex.hIcon           = LoadIcon(hInstance,
11                               (LPCTSTR)IDI_WINAPPSAMPLE);//图标
12      wcex.hCursor         = LoadCursor(NULL, IDC_ARROW); //光标
13      wcex.hbrBackground   = (HBRUSH)(COLOR_WINDOW+1);    //背景颜色
14      wcex.lpszMenuName    = (LPCSTR)IDC_WINAPPSAMPLE;    //菜单
15      wcex.lpszClassName   = szWindowClass;               //窗体类名
16      wcex.hIconSm         = LoadIcon(wcex.hInstance,
17                            (LPCTSTR)IDI_SMALL);          //小图标
18      return RegisterClassEx(&wcex);                      //注册窗体类
19  }
```

上面的代码注册 szWindowClass 变量指定的窗体对应的窗体类。RegisterClassEx()函数的参数是 WNDCLASSEX 结构，存储了窗体的属性。

5.2.3　使用 CreateWindow()创建窗口

注册完窗体类后，就需要调用 CreateWindow()函数创建窗口。CreateWindow()函数用于创建已经注册了的窗体类的窗口。第一个参数是注册的窗口类的名称，其余的参数指定了窗口的其他属性。调用完此函数后，需要判断创建是否成功。如果创建成功，则调用 ShowWindow()函数显示窗口。下面是创建窗口代码的封装。

```
01  //初始化实例
02  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
03  {
04     HWND hWnd;                              //定义窗口句柄
05     hInst = hInstance;                      //在全局变量 hInst 中存储实例句柄
06     hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
07     CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
08                                             //创建窗口
09     if (!hWnd)
10       return FALSE;                         //创建失败，则返回
11     ShowWindow(hWnd, nCmdShow);             //显示窗口
12     UpdateWindow(hWnd);                     //更新窗口
13     return TRUE;                            //返回
14  }
```

上面的代码中，在 InitInstance()函数中调用 CreateWindow()函数创建窗口，如果创建窗口成功，则调用 ShowWindow()函数显示窗口。

5.2.4　使用消息循环响应用户输入

创建窗口后，虽然窗口显示在界面上，但是，此时窗口并不能响应用户输入的任何命

令。要使窗口响应用户的输入，需要让窗口执行消息循环。为此在 VC 中，一旦主窗口创建并显示后，WinMain()函数可以进入主任务，从应用程序队列中读取消息，并分配给相应的窗口。Windows 不能直接发送输入给应用程序，而会将所有的鼠标和键盘输入消息发送到消息队列中。应用程序必须从消息队列中读取消息和接收消息，并将消息分配给窗口函数，根据消息类型进行处理。代码如下：

```
01  while (GetMessage(&msg, NULL, 0, 0))                //主消息循环
02  {
03      //转换消息快捷键
04      if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
05      {
06          TranslateMessage(&msg);                     //转换消息
07          DispatchMessage(&msg);                      //调度消息
08      }
09  }
```

其中，GetMessage()函数从队列中读取消息。TranslateAccelerator()函数翻译快捷键消息。如果消息按键在快捷键列表中存在，则执行快捷键对应的命令。TranslateMessage()函数会将按键消息转换成字符消息，而 DispatchMessage()函数则将消息发送给相应的窗口函数。

5.2.5　主窗体函数 WinProc()

每个窗体都有一个窗体函数，并且可以在注册窗体类时，在 WNDCLASSEX 结构的 lpfnWndProc 成员中指定窗体函数的名称。使用向导生成的 Win32 程序的主窗体函数如下：

```
LRESULT CALLBACK WndProc(
    HWND     hWnd,
    UINT     message,
    WPARAM   wParam,
    LPARAM   lParam
)
```

其中，CALLBACK 修饰符用于指定函数使用标准函数调用转换。对话框程序接收的消息，可能是输入消息，也可能是从系统发送的窗体管理消息。程序员可以在窗体函数中有选择地处理感兴趣的消息，或采取默认处理，通过调用 DefWindowProc()函数将消息传给窗体。代码如下：

```
01  switch (message)                           //根据消息类型执行相应的操作
02  {
03      case WM_COMMAND:                       //如果是 WM_COMMAND 类型的
04          wmId    = LOWORD(wParam);          //获取发送命令的对象 ID
05          wmEvent = HIWORD(wParam);          //获取发送的事件
06          switch (wmId)                      //解析菜单选择
07          {
08              case IDM_ABOUT:                //如果是“关于”命令，则显示“关于”对话框
09                  DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX,
10                                        hWnd, (DLGPROC)About);
11                  break;
12              case IDM_EXIT:                 //如果是“退出”命令，则退出窗体
13                  DestroyWindow(hWnd);
```

```
14                  break;
15              default:                    //其他命令，则做相应处理
16                  return DefWindowProc(hWnd, message, wParam, lParam);
17          }
18          break;
19
20      case WM_PAINT:                      //如果是重绘命令
21          hdc = BeginPaint(hWnd, &ps);//开始重绘
22          RECT rt;                        //定义区域变量
23          GetClientRect(hWnd, &rt);       //获取客户区范围
24          DrawText(hdc, szHello, strlen(szHello), &rt, DT_CENTER);
25                                          //绘制欢迎文本
26          EndPaint(hWnd, &ps);            //结束重绘
27          break;
28
29      case WM_DESTROY:                    //销毁窗体
30          PostQuitMessage(0);             //发送退出消息
31          break;
32
33      default:                            //默认情况，处理窗体消息
34          return DefWindowProc(hWnd, message, wParam, lParam);
35  }
```

在上面代码中，switch 语句，首先判断消息是否为 WM_COMMAND 消息，表示用户从菜单中选择了菜单项。在此部分中，又使用了一条 switch 语句，判断调用了哪个菜单项。如果选择了 About 菜单项，则弹出自定义的“关于”对话框。如果选择了退出命令，则销毁此窗体。否则，使用默认的 DefWindowProc()消息处理函数进行处理。

而 switch 语句的第二条 case 语句，是处理 WM_PAINT 消息，此消息表示程序需要重绘应用程序中的部分或所有的窗体。使用 BeginPaint()函数获取设备上下文句柄，使用 DrawText()等函数在应用程序窗体中重绘。重绘完成后，使用 EndPaint()函数释放设备上下文。此例子重绘时，在对话框中显示当前 szHello 变量代表的字符串。

大部分窗体程序都需要处理 WM_DESTROY 消息，此消息用于通知窗体要销毁了。收到此消息时，窗体程序会发送 WM_QUIT 消息到应用程序的消息队列中。在此例中，其他消息使用默认的 DefWindowProc()消息处理函数进行处理。

5.2.6　Windows 编程实例——设计一个电子时钟

本小节改写 4.8 节中的电子时钟例子，将其改写为 Windows 窗口程序。代码如下：

```
01  #include "stdafx.h"                         //引用 stdafx 头文件
02  #include "resource.h"                       //引用资源文件
03  #include <time.h>                           //引用 time 头文件
04
05  #define MAX_LOADSTRING 100                  //定义装载字符串的最大长度
06  #define WM_TIMER_CLOCK  WM_USER + 20        //定义时钟定时器消息
07  #define WM_CLOCK_INTERVAL   1000            //定义时钟刷新时间间隔
08
09  HINSTANCE   hInst;                          //当前实例句柄
10  TCHAR       szTitle[MAX_LOADSTRING];        //标题栏文本
11  TCHAR       szWindowClass[MAX_LOADSTRING];  //标题栏文本
12  struct tm * newdate;                        //当前时间
```

```
13  time t      long date;                        //时间描述
14  TCHAR szTimerTitle[MAX_LOADSTRING];           //定时器文本
15
16  //此模块中包含的函数声明
17  ATOM                 MyRegisterClass(HINSTANCE hInstance);//注册窗体类
18  BOOL                 InitInstance(HINSTANCE, int);        //初始化实例
19  LRESULT CALLBACK     WndProc(HWND, UINT, WPARAM, LPARAM); //窗体处理函数
20  LRESULT CALLBACK     About(HWND, UINT, WPARAM, LPARAM);   //关于函数
21
22  //主函数
23  int APIENTRY WinMain(HINSTANCE hInstance,
24      HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
25  {
26      MSG msg;                                             //定义消息
27      HACCEL hAccelTable;                                  //定义加速键变量
28      LoadString(hInstance, IDS APP TITLE, szTitle, MAX LOADSTRING);
29      //装载应用程序标题
30      LoadString(hInstance, IDC_WINAPPSAMPLE, szWindowClass,
31                                                    MAX_LOADSTRING);
32      MyRegisterClass(hInstance);                          //注册应用程序类
33      if (!InitInstance (hInstance, nCmdShow))
34          return FALSE;
35      //完成应用程序初始化
36      hAccelTable = LoadAccelerators(hInstance,
37                              (LPCTSTR)IDC_WINAPPSAMPLE);//装载加速键
38      while (GetMessage(&msg, NULL, 0, 0))                 //主消息循环
39      {
40          //转换快捷键消息
41          if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
42          {
43              TranslateMessage(&msg);                      //转换消息
44              DispatchMessage(&msg);                       //调度消息
45          }
46      }
47      return msg.wParam;                                   //返回消息的参数
48  }
49  ATOM MyRegisterClass(HINSTANCE hInstance)                //注册窗体类
50  {
51      ...                                                  //与前面定义相同
52  }
53  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)     //初始化实例
54  {
55      ...                                                  //与前面定义相同
56      //创建定时器
57      SetTimer(hWnd, WM TIMER CLOCK, WM CLOCK INTERVAL, NULL);
58      return TRUE;
59  }
60  LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
61                  WPARAM wParam, LPARAM lParam)            //消息处理函数
62  {
63      ...
64          switch (message)
65      {
66          ...
67          case WM_TIMER:                                   //加入定时器处理
68              if (wParam == WM TIMER CLOCK)
69                  //传入的定时器类型为显示时间定时器
```

```
70              {
71                  time( &long_date );                       //获取时间
72                  newdate = localtime( &long_date );        //转换成本地时间
73                  memset(szTimerTitle, 0, sizeof(szTimerTitle));
74                  //打印当前时间
75                  strcpy( szTimerTitle, asctime( newdate ) );
76                  UpdateWindow(hWnd);                       //显示窗口
77              }
78              break;
79              ...
80          }
81          return 0;
82      }
83      //关于对话框的消息处理函数
84      LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam,
85                                                            LPARAM lParam)
86      {
87          switch (message)
88          {
89          case WM_INITDIALOG:
90              return TRUE;
91          case WM_COMMAND:
92              if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
93              {
94                  //退出对话框
95                  EndDialog(hDlg, LOWORD(wParam));
96                  return TRUE;
97              }
98              break;
99          }
100         return FALSE;
101     }
```

本示例中定义了类用到的全局变量，实现了 WinMain()程序入口函数，注册了对话框类 MyRegisterClass 并实现了 InitInstance()初始化实例函数和 WndProc()对话框函数，同时实现了关于对话框的使用。因为本节是介绍 Windows 程序，所以没有将所有代码列出来，在后面为了节约篇幅，同样不再显示 IDE 自动生成的代码和不相关的代码。

5.3　MFC 基础

MFC 是 VC 非常重要的组成部分，它为开发人员封装了很多常用的功能类。如果能够熟练使用 MFC 中的类，则可以快速提高开发效率。因为有些功能是对底层 Windows API 的封装，所以，熟悉 Windows API 编程的开发人员，可以将 MFC 与对应的 Windows API 函数结合起来学习，会达到事半功倍的效果。本节将介绍 MFC 基础。

5.3.1　什么是微软基础类库 MFC

微软基础类库（Microsoft Foundation Class Library，MFC）是一个编写 Windows 应用程序的框架类库。使用 MFC 类库编写 C++程序，可以便捷地实现界面功能、网络功能、多媒体功能和数据访问功能等各种常用功能。MFC 不仅可以将需要增加的功能代码添加到

框架代码中，而且还提供了对 C++类特性的支持，因此使用 MFC 可以扩展或重写 MFC 框架提供的基本功能。

- MFC 框架是能完成 Windows 核心编程的有效的框架类库。
- 使用 MFC 框架，可以缩短开发时间，使得代码更简洁，并且可以在不减少程序开发自由度和灵活性的条件下，完成多种功能的开发。
- MFC 能提供更高级的编程接口技术，如 Active 技术、OLE 和 Internet 编程等。
- MFC 通过 DAO 和 ODBC 简化了数据库编程，通过 Windows Sockets 简化了网络编程。
- MFC 还提供了属性页、打印预览、浮动工具栏和自定义工具栏等常用的编程属性。

使用 MFC 框架编程主要是基于一组类和几个工具。一部分类封装了 Win32 应用程序编程接口（API）的很多功能，另一部分类实现了应用程序架构，如文档类、视图类和应用程序框架类等。还有一部分类封装了 OLE 特性和 ODBC 和 DAO 的数据访问功能，如 MFC 的 CWnd 类封装了 Win32 的窗体概念。也就是说，C++类 CWnd 封装了 HWND 句柄，此句柄表示一个 Windows 窗体。同样，CDialog 类封装了 Win32 的对话框。

封装的含义就是，如 C++类 CWnd 包含一个 HWND 类型的成员变量，类的成员函数封装了使用 HWND 作为参数的 Win32 函数。通常类成员函数与封装的 Win32 函数具有相同的名称。

5.3.2　MFC 类层次结构

MFC 基础类覆盖了多种功能，包括支持应用程序框架的类、支持窗体的类 CWnd、数据库类、Socket 类和文件类等。如图 5-1 所示为 MFC 的层次结构。

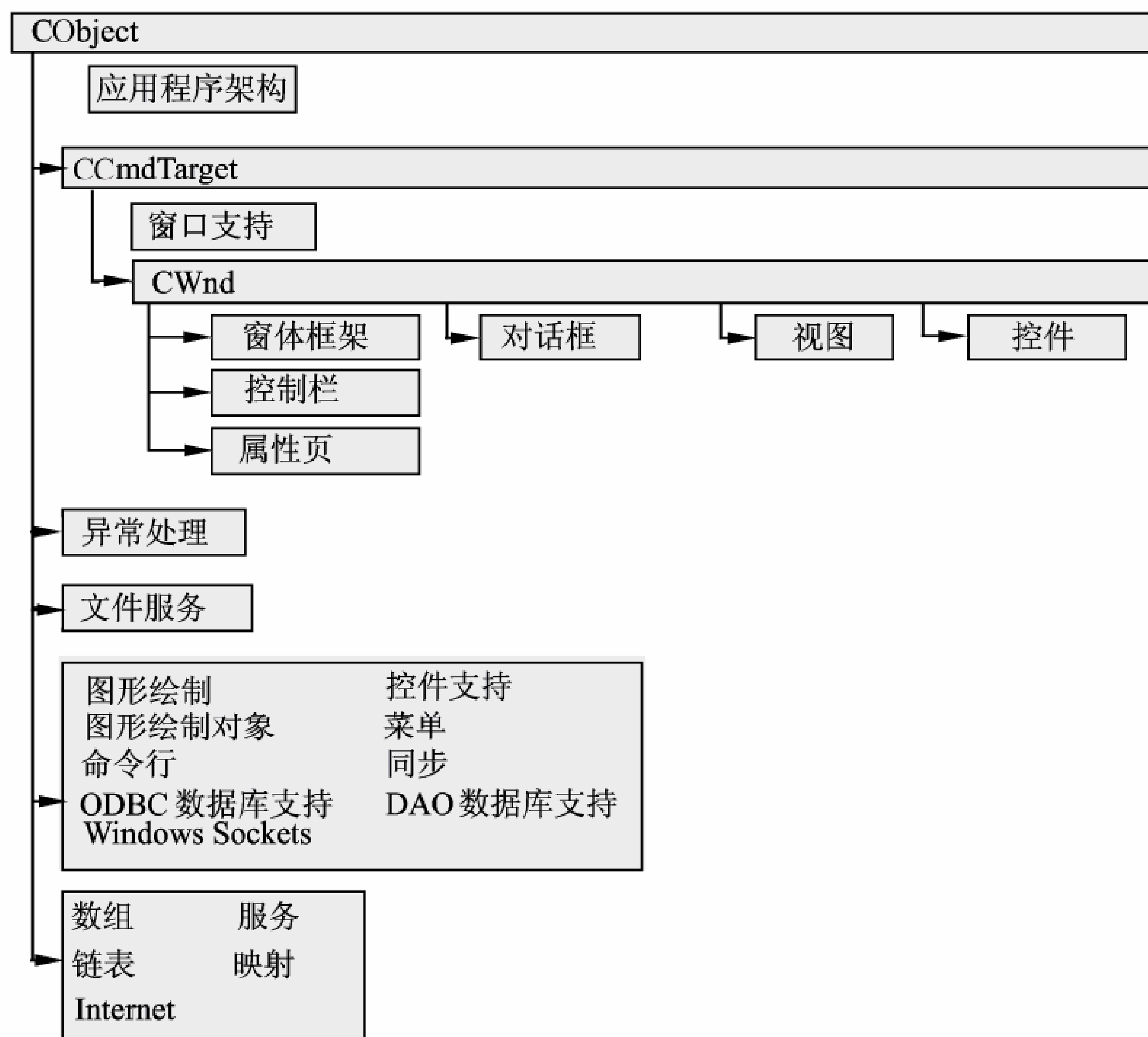


图 5-1　MFC 的层次结构

其中，CObject 类是所有类的基类。在此基础之上，派生了 CCmdTarget 类，用于表示命令目标对象，CWnd 类就是继承于此类。而在 CWnd 类上又派生了包括对话框、视图、控件、属性页、窗体框架和控制栏等在内的扩展对话框类。

基于 CObject 类，还实现了异常处理类和文件服务类，并实现了对图形控制、控件支持、图形绘制对象、菜单、命令行、数据库支持、同步和 Windows Socket 的支持。还包括常用的数组、链表、映射和 Internet 服务的 MFC 类的实现。

5.3.3　MFC 全局函数

MFC 中除了 MFC 类外，还包括部分宏和全局成员，这些都不属于类的成员，比如全局函数和全局变量。全局函数涉及的方面非常广，包括数据类型、类型转换、运行时对象服务模型、诊断服务、异常处理、CString 的格式化和消息对话框的显示、消息映射、应用程序信息和管理、标准命令和窗口标识、集合类等。常用的 MFC 全局函数如表 5-3 所示。

表 5-3　常用的MFC全局函数

全 局 函 数	功　　能
AfxAbort()	MFC 提供的默认终止函数
AfxBeginThread()	创建新线程
AfxCheckError()	检测代码是否为错误代码
AfxCheckMemory ()	检测是否发生有关内存的错误
AfxDaoInit()	初始化 DAO 数据库引擎
AfxDaoTerm()	终止 DAO 数据库引擎
AfxDbInitModule()	初始化 MFC 数据库 DLL
AfxDoForAllClasses()	在应用程序内存空间中，枚举所有序列化派生类
AfxDump()	调试程序时，列出对象所有的状态
AfxDumpStack()	列出当前堆栈的情况
AfxEnableControlContainer()	支持对 OLE 控件的支持
AfxEnableMemoryTracking()	打开内存跟踪
AfxEndThread()	结束线程
AfxFreeLibrary()	释放对 DLL 的引用
AfxGetApp()	获取应用程序对象
AfxGetAppName()	获取应用程序名称
AfxGetHENV()	获取当前使用的 ODBC 句柄
AfxGetInstanceHandle()	获取当前应用程序的实例句柄
AfxGetInternetHandleType()	获取 Internet 句柄类型
AfxGetMainWnd()	获取应用程序主对话框
AfxGetResourceHandle()	获取资源句柄
AfxGetStaticModuleState()	获取静态模块状态
AfxGetThread()	获取当前执行的线程
AfxInitExtensionModule()	初始化 DLL
AfxInitRichEdit()	初始化应用程序的编辑框

续表

全 局 函 数	功　　能
AfxIsMemoryBlock()	判断指定内存块是否是有效的内存空间
AfxIsValidAddress()	判断是否是有效的内存地址
AfxIsValidString()	判断是否是有效的字符串
AfxLoadLibrary()	装载 DLL
AfxMessageBox()	调用消息对话框
AfxNetInitModule()	初始化 MFC 的 Socket DLL
AfxOleCanExitApp()	判断 OLE 是否可以退出
AfxParseURL()	解析 URL 地址
AfxRegisterClass()	在 DLL 中注册对话框类
AfxRegisterWndClass()	MFC 自动注册几个有用的对话框类
AfxSetAllocHook()	在每次分配内存时，允许设置钩子函数
AfxSetResourceHandle()	设置资源句柄
AfxSocketInit()	初始化对 Windows Socket 的支持
AfxThrowDaoException()	抛出 DAO 异常
AfxThrowDBException()	抛出 CDBException 类型的异常
AfxThrowFileException()	抛出文件异常
AfxThrowInternetException()	抛出 Internet 异常
AfxThrowMemoryException()	抛出内存异常
AfxThrowNotSupportedException()	抛出不支持的异常
AfxThrowOleDispatchException()	抛出 OLE 调度异常
AfxThrowOleException()	抛出 OLE 异常
AfxThrowResourceException()	抛出资源异常
AfxThrowUserException()	抛出用户异常
AfxWinInit()	初始化对话框应用程序

5.4　MFC 应用程序框架分析

要熟练掌握 MFC 应用程序的开发，首先需要了解 MFC 应用程序框架结构，熟悉其中的各个元素及预定义的处理函数，了解 MFC 应用程序的运行过程，并能快速定位到功能代码处。本节就来分析 MFC 应用程序框架。

5.4.1　MFC 的入口函数 WinMain()

MFC 中主应用程序类封装了初始化、运行和终止 Windows 应用程序的功能。基于 MFC 框架的应用程序必须具有一个派生自 CWinApp 类的对象。此对象在创建窗体前进行初始化。而 CWinApp 类派生自 CWinThread 类，代表应用程序执行的主线程，但是一个应用程序可能具有一个或多个线程。CWinThread 类中具有 InitInstance()、Run()、ExitInstance()和 OnIdle()成员函数。这些函数在 CWinApp 中也被重载使用，但是它们是作为应用程序对象执行而不是主线程。

像其他 Windows 程序一样，MFC 应用程序也具有一个 WinMain()入口函数。在 MFC 应用程序中，不需要重写 WinMain()函数，由类库提供，并且在应用程序启动时调用。WinMain()函数完成诸如注册对话框类等标准服务，然后调用应用程序对象的初始化成员函数，并运行程序。当然，也可以根据需要重写 CWinApp 的 WinMain()成员函数。

WinMain()函数调用应用程序对象的 InitApplication()和 InitInstance()成员函数初始化应用程序，调用 Run()成员函数运行应用程序的消息循环，调用应用程序对象的 ExitInstance()成员函数退出应用程序。图 5-2 演示了执行 MFC 应用程序的过程。

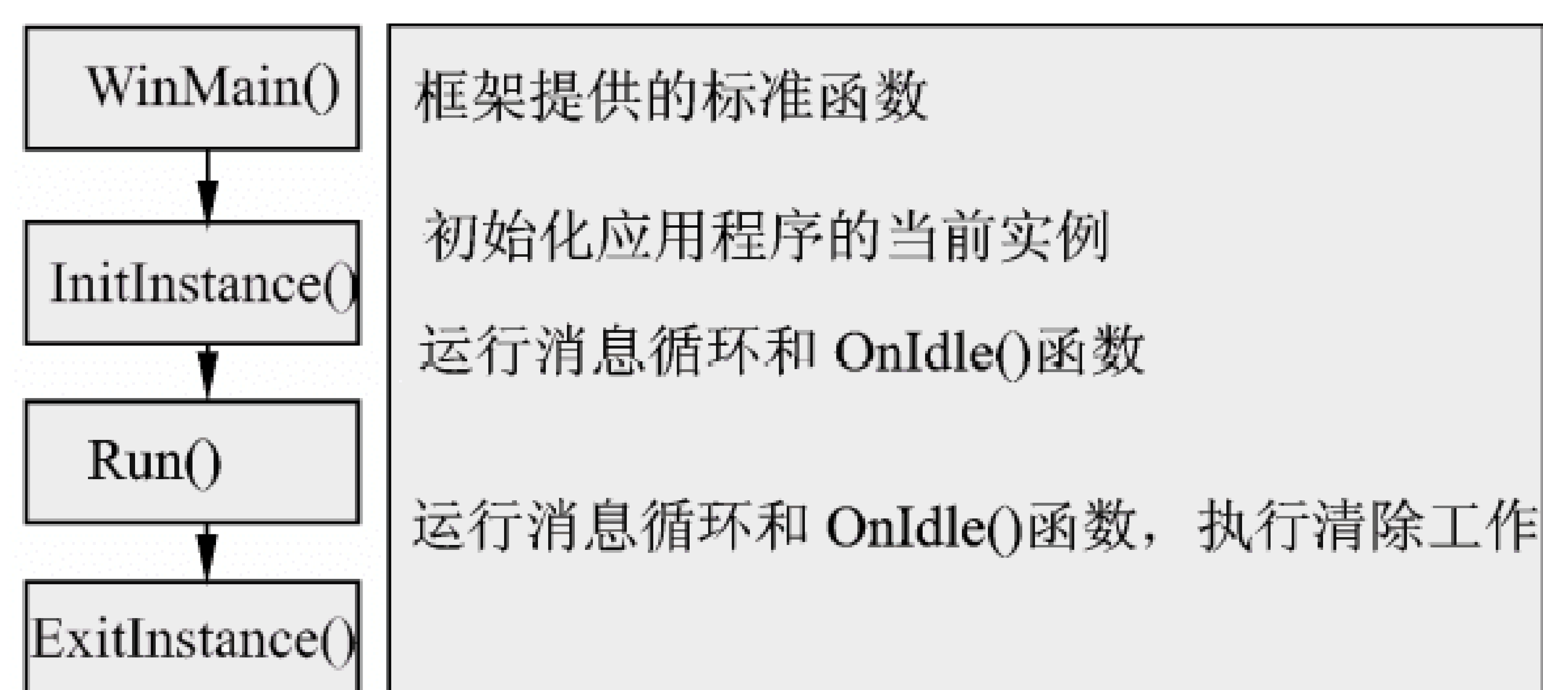


图 5-2　WinMain()函数的处理流程

5.4.2　派生自 CWinApp 的应用程序对象

CWinApp 类是从 Windows 应用程序对象中派生而来的应用程序基类。应用程序对象提供初始化应用程序和实例的成员函数，并提供运行应用程序和终止应用程序的函数。每个使用 MFC 的应用程序只能包含一个派生自 CWinApp 的对象。当构造完 C++全局对象后，框架会构造此对象，因此，当 Windows 调用 MFC 类库提供的 WinMain()函数时，应用程序对象已经有效了。MFC 应用程序在全局级别上声明派生自 CWinApp 类的应用程序对象，并使用此对象进行应用程序的相关操作。

当从 CWinApp 类派生应用程序类，重写 InitInstance()成员函数创建应用程序的主对话框对象时，除了 CWinApp 的成员函数，MFC 类库提供了下面的全局函数访问 CWinApp 对象和其他全局信息。

- ❑ AfxGetApp()函数：获取应用程序的 CWinApp 对象的指针。
- ❑ AfxGetInstanceHandle()函数：获取当前应用程序实例的句柄。
- ❑ AfxGetResourceHandle()函数：获取当前应用程序的资源句柄。
- ❑ AfxGetAppName()函数：获取包含应用程序名称的字符串指针。如果获取的是 CWinApp 对象的指针，那么可以通过它的 m_pszExeName 成员获取应用程序名称。

5.4.3　初始化应用程序的 InitInstance()函数

Windows 允许同时运行同一个程序的一个或多个实例。WinMain()函数在每次启动新的应用程序实例时，会调用 InitInstance()函数。应用向导创建的标准的 InitInstance()函数主要完成以下工作。

- ❑ 创建文档模板，依次创建文档对象、视图对象和框架对话框。

- ❑ 从.INI 文件中或 Windows 注册表中装载标准文件选项，包括最近使用的文件名称等信息。
- ❑ 注册一个或多个文档模板。
- ❑ 对于 MDI 应用程序，创建一个主框架对话框。
- ❑ 在命令行上处理命令行打开文档，或打开新的空文档。

开发人员可以向其中添加自定义的初始化代码或修改向导编写的代码。下面是使用应用向导创建的 InitInstance()函数的代码。

```
01  BOOL CWinMFCSampleApp::InitInstance()        //初始化应用程序实例
02  {
03      AfxEnableControlContainer();             //加入控件包容器功能
04  #ifdef _AFXDLL                               //判断是否定义了 AFXDLL
05      Enable3dControls();
06  #else
07      Enable3dControlsStatic();
08  #endif
09      //设置注册表键
10      SetRegistryKey(_T("Local AppWizard-Generated Applications"));
11      LoadStdProfileSettings();                //装载标准配置设置
12      CMultiDocTemplate* pDocTemplate;         //定义多文档模板变量
13      pDocTemplate = new CMultiDocTemplate(IDR_WINMFCTYPE,
14          RUNTIME_CLASS(CWinMFCSampleDoc),
15          RUNTIME_CLASS(CChildFrame),
16          RUNTIME_CLASS(CWinMFCSampleView)); //创建多文档模板
17      AddDocTemplate(pDocTemplate);            //增加到文档模板集合中
18      CMainFrame* pMainFrame = new CMainFrame;//定义 CMainFrame 类
19      //装载框架
20      if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
21          return FALSE;
22      m_pMainWnd = pMainFrame;                 //为主窗口类赋值
23      CCommandLineInfo cmdInfo;                //定义命令行信息变量
24      ParseCommandLine(cmdInfo);               //解析命令行
25      //处理命令行
26      if (!ProcessShellCommand(cmdInfo))
27          return FALSE;
28      pMainFrame->ShowWindow(m_nCmdShow);      //显示主窗口
29      pMainFrame->UpdateWindow();              //刷新主窗口
30      return TRUE;
31  }
```

上面的代码主要完成了以下工作。

- ❑ 调用 AfxEnableControlContainer()函数开启 OLE 包容器功能。
- ❑ 判断是否预定义了_AFXDLL，如果定义了，表示使用共享版本的 DLL；如果没有定义，则表示使用静态版本的 MFC DLL，分别调用 Enable3dControls()函数和 Enable3dControlsStatic()函数，表示开启 3D 显示和 3D 静态显示。
- ❑ 调用 SetRegistryKey()函数存储注册表键。
- ❑ 调用 LoadStdProfileSettings()函数装载标准文件选项。
- ❑ 创建 CMultiDocTemplate 对象，注册一个或多个文档模板。
- ❑ 创建 CMainFrame 对象，创建主对话框。
- ❑ 创建 CCommandLineInfo 对象，创建命令行信息，执行命令行解析并进行处理。

- 调用 pMainFrame 的 ShowWindow()函数和 UpdateWindow()函数，显示主对话框并刷新界面显示。

5.4.4　框架程序的运行核心 Run()函数

框架应用程序运行时就是运行 CWinApp 类的 Run()函数。初始化后，WinMain()函数调用 Run()函数处理消息循环。Run()通过消息循环，检查消息队列中的有效消息。如果消息有效，Run()会根据消息类型的不同采取不同的处理方式。如果没有消息可用，Run()函数就调用 OnIdle()函数完成空闲时程序或框架需要执行的操作。程序大部分情况下是空闲的，只有当消息到达时，进程才会处理这些消息。如果没有消息也没有空闲处理要做，应用程序会一直等待，直到需要处理时，才会执行操作。当终止应用程序时，Run()函数会调用 ExitInstance()函数。

5.5　MFC 的消息映射

在 Windows 系统中，消息一般由从 CWnd 派生而来的对象处理，包括 CFrameWnd、CMDIFrameWnd、CMDIChildWnd、CView、CDialog 和其他从这些类派生而来的对象。这些对象封装了代表 Windows 窗口句柄的 HWND。

在传统的 Windows 程序中，MFC 使用 switch 语句处理发送给窗口的消息，在窗口类中定义消息到成员函数之间的映射，当窗口处理消息时，会自动地调用相应的成员函数。VC 中使用消息映射需要执行下面几个步骤。

（1）在头文件中使用 DECLARE_MESSAGE_MAP 宏声明消息映射，放在类声明的结束部分。

（2）在源文件中，使用 BEGIN_MESSAGE_MAP 宏和 END_MESSAGE_MAP 宏定义消息映射，消息映射必须定义在函数和类定义外的地方。

（3）在头文件中，使用 AFX_MSG 宏声明消息函数。

（4）在源文件中重载或新定义消息函数的实现代码。

5.5.1　标准 Windows 消息

为了简化工作，Windows 系统提供了一组标准 Windows 消息，一般由对话框类和视图类根据参数的取值进行处理。比如，创建窗口、销毁窗口和窗口重绘等。每个标准 Windows 消息都有一个以 WM 开头的消息 ID 和对应的宏，格式是 ON_WM_xxx，其中 xxx 是消息名称，例如 ON_WM_CREATE 宏表示创建对话框消息。标准 Windows 消息对应的处理函数名根据消息宏派生而来，格式是 OnXxx，其中 Xxx 与消息宏中的 xxx 是一致的，只是单词的第一个字母为大写。消息处理函数的参数顺序依次是 wParam 和 lParam。以下代码演示了 MFC 如何实现标准 Windows 消息映射。在类的头文件中，代码如下：

```
01  class CMainFrame : public CMDIFrameWnd
02  {
```

```
03  protected:
04      //{{AFX_MSG(CMainFrame)
05      afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
06      //}}AFX_MSG
07      DECLARE_MESSAGE_MAP()
08  };
```

在类的实现文件中，代码如下：

```
01  BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
02      //{{AFX_MSG_MAP(CMainFrame)
03      ON_WM_CREATE()
04      //}}AFX_MSG_MAP
05  END_MESSAGE_MAP()
```

上面代码显示了 WM_PAINT 消息映射的实现。首先在头文件中类声明的结尾处使用 DECLARE_MESSAGE_MAP 宏声明消息映射。其次，在源文件中，使用 BEGIN_MESSAGE_MAP 宏和 END_MESSAGE_MAP 宏定义 ON_WM_CREATE 消息映射，并且在头文件类声明的 AFX_MSG 宏之间定义 ON_WM_CREATE 消息的 OnCreate()消息处理函数。最后在源文件中定义消息函数的实现。

5.5.2　触发菜单/快捷键产生的命令消息

MFC 除了支持标准 Windows 消息外，还支持命令消息。命令消息是指当用户触发菜单或快捷键时发送的消息。使用 ON_COMMAND 宏可以在消息映射表中指定命令消息对应的处理函数。使用 ON_UPDATE_COMMAND_UI 宏可以在消息映射表中指定命令更新消息对应的处理函数。宏的第一个参数是命令 ID，命令 ID 就是指在定义菜单项或快捷键时使用的控件 ID，第二个参数是命令消息的处理函数。命令处理函数没有参数和返回值，命令更新处理函数只有一个 CCmdUI 类型的参数并且没有返回值。其定义方式为：

```
ON_COMMAND(id, memberFxn)                    //命令消息宏
ON_UPDATE_COMMAND_UI(id, memberFxn)          //命令更新消息定义
```

以下代码演示了如何使用 ON_COMMAND 宏和 ON_UPDATE_COMMAND_UI 宏处理命令消息。在类的定义文件中，添加代码如下：

```
01  #define    ID_MYCMD     100
02  afx_msg void OnMyCommand();
03  afx_msg void OnUpdateMyCommand(CCmdUI* pCmdUI);
```

在类的实现文件中，添加代码如下：

```
01  //在消息映射定义中
02  ON_COMMAND(ID_MYCMD, OnMyCommand)
03  ON_UPDATE_COMMAND_UI(ID_MYCMD, OnUpdateMyCommand)
04  //在实现文件中
05  void CMyClass::OnMyCommand()                        //命令处理函数
06  {
07      ...
08  }
09  void CMyClass::OnUpdateMyCommand(CCmdUI* pCmdUI)    //通过 pCmdUI 设置 UI
10  {
```

```
11      ...
12  }
```

上面代码演示了 ID_MYCMD 命令的消息处理实现和命令更新消息的实现。

5.5.3 使用 ON_MESSAGE 宏自定义消息

MFC 除了支持 Windows 系统消息外，还支持用户自定义消息。使用 ON_MESSAGE 宏可以在消息映射表中指定消息对应的处理函数。代码如下：

```
01  #define WM_MYMESSAGE    WM_USER + 100           //自定义消息值
02  //自定义消息处理函数
03  afx_msg LRESULT OnMyMessage(WPARAM wParam, LPARAM lParam);
04  //在类的实现文件中，自定义消息映射函数
05  BEGIN_MESSAGE_MAP(CMyWnd, CMyParentWndClass)
06      ON_MESSAGE(WM_MYMESSAGE, OnMyMessage)
07  END_MESSAGE_MAP()
```

上面代码中第一条语句使用#define 定义了自定义消息 ID 值。第二条语句定义了自定义消息的处理函数。下面的代码在消息映射表中，指定自定义消息 WM_MYMESSAGE 的处理函数为 OnMyMessage()函数。定义好自定义消息及其处理函数后，就可以在程序的其他地方发送自定义消息，代码如下：

```
CWnd* pWnd = ...;                           //定义窗口变量
pWnd->SendMessage(WM_MYMESSAGE);            //发送自定义消息
```

上面代码向窗口发送自定义 WM_MYMESSAGE 消息，窗口接收到消息后会调用 OnMyMessage()函数进行消息处理。要注意的是，用户自定义消息的消息 ID 值的范围是从 WM_USER～0x7fff。

5.5.4　注册系统消息

上面这 3 种消息都是基于同一个窗口下的消息处理。要在系统中定义一个独立于窗口的唯一的消息处理，可以使用 Windows 注册消息。使用 RegisterWindowMessage()函数可以创建在系统中唯一的消息 ID。使用 ON_REGISTERED_MESSAGE 宏可以在消息映射表中指定 Windows 注册消息对应的处理函数，宏的参数为使用 RegisterWindowMessage()函数返回的 UINT 类型的消息 ID。代码如下：

```
01  //{{AFX_MSG(CWinMFCSampleDoc)                   //文档类中的消息处理函数
02  afx_msg LRESULT OnParse(WPARAM wParam, LPARAM lParam);
03                                                  //注册消息处理函数
04  //}}AFX_MSG
05  //注册 Windows 消息
06  static UINT NEAR WM_PARSE = RegisterWindowMessage("COMMDLG_PARSE");
07  BEGIN_MESSAGE_MAP(CWinMFCSampleDoc, CDocument)
08  //{{AFX_MSG_MAP(CWinMFCSampleDoc)
09      ON_REGISTERED_MESSAGE(WM_PARSE, OnParse)    //消息映射处理函数
10  //}}AFX_MSG_MAP
11  END_MESSAGE_MAP()
```

上面代码使用 RegisterWindowMessage()函数注册了名称为 COMMDLG_PARSE 的消息，消息 ID 存储在 WM_PARSE 变量中，并定义了此消息的处理函数为 OnParse。Windows 注册消息的 ID 范围值是从 0xC000～0xFFFF。从上面可以看出，使用 Windows 注册消息可以实现通过消息完成进程间通信，只要进行通信的进程定义了相同名称的消息即可。有关此方面的应用在后面的章节会介绍。

5.6　本 章 小 结

本章为 Windows 编程的引入章，主要引入了 Windows 编程和 MFC 基础。因为 VC 主要是编写 Windows 应用程序，所以对 Windows 编程具有良好的支持，并在其基础上进行了封装。经过这层封装，将 Windows 底层的编程与用户编程之间良好地联系起来，使得开发人员使用 MFC 的类库就可以快速地编写 Windows 应用程序。本章的难点在于透彻理解 MFC 应用程序的框架，为后面进行 Windows 开发打好基础。从第 6 章开始，将介绍有关 Windows 程序的界面开发。

5.7　习　　题

1．使用 Visual Studio 2010 提供的程序模板生成一个 Win 32 应用程序，什么代码都不添加的时候程序运行的效果如图 5-3 所示。应用本章 5.2 节所学到的知识试着分析一下这个由向导生成的程序。

【思路】分析一个程序的时候，可以按照程序的执行流程来分析，就像 5.2 节所讲的那样。

2．使用 Visual Studio 2010 提供的程序模板生成一个 MFC 的多文档应用程序。不对向导的选项进行修改，那么由向导生成的多文档的应用程序应该是如图 5-4 所示的样子。应用本章 5.3～5.5 节所学到的知识，试着分析一下这个由向导生成的程序。

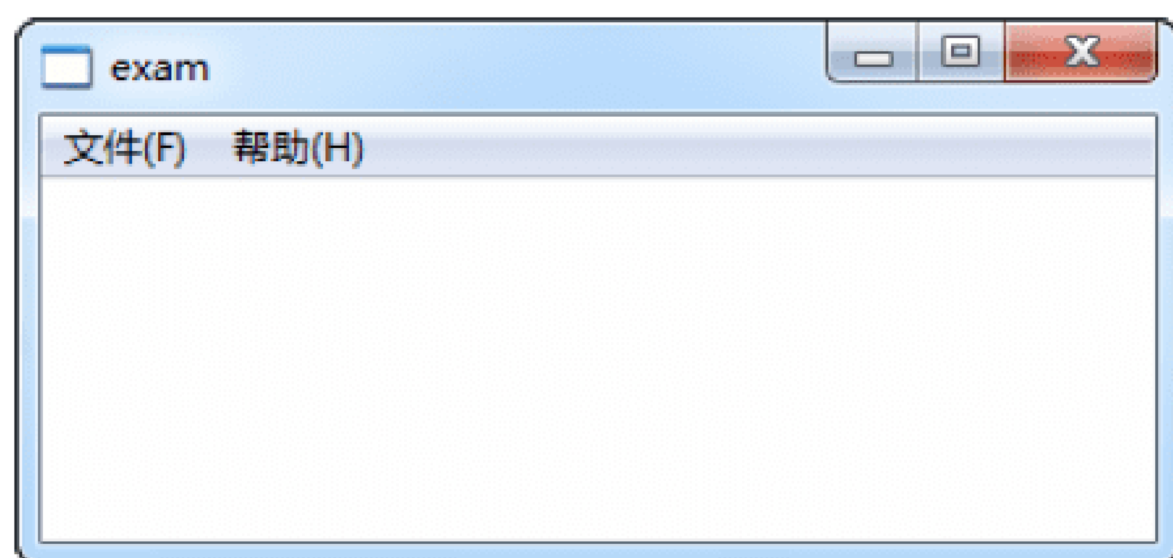


图 5-3　向导生成的 Win 32 程序的运行效果

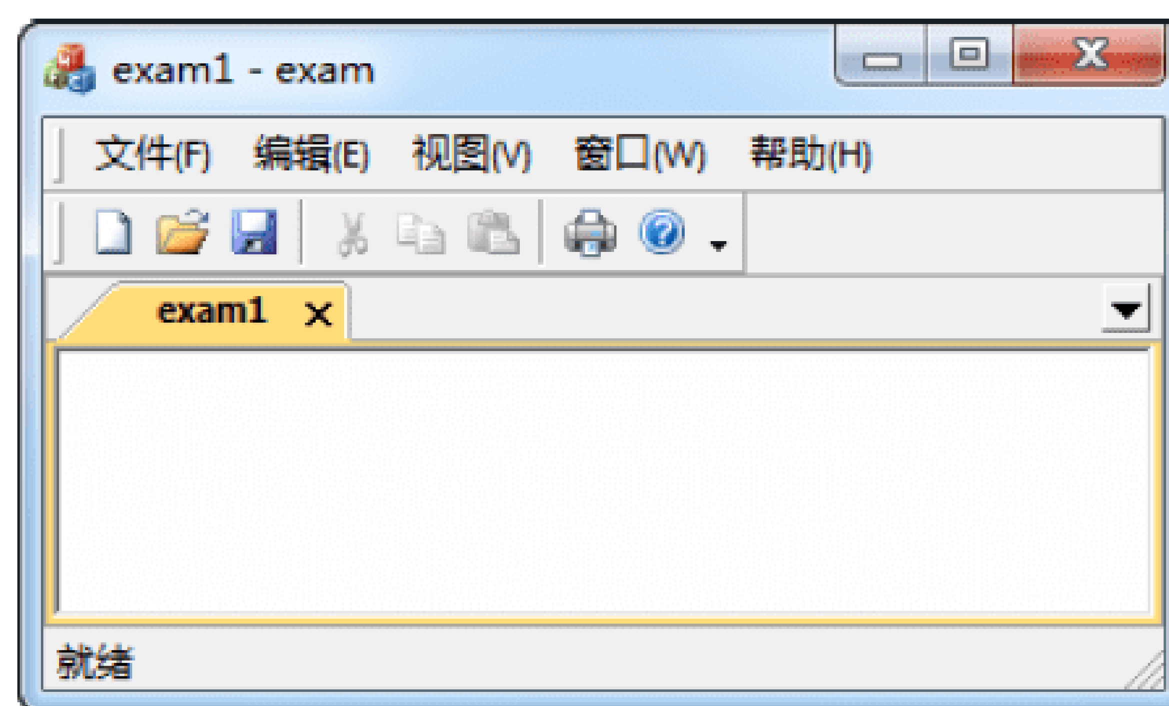


图 5-4　向导生成的 MFC 多文档程序的运行效果

【思路】同上一题类似，本题也要求分析一个程序，方法也是类似的，需要按照 5.4 节所讲的知识依次对应到向导所生成的程序中即可。

第 6 章　菜单、工具栏和状态栏

菜单是管理菜单命令的控件，工具栏是管理工具按钮的控件，菜单的菜单项和工具栏的按钮可以关联起来。状态栏用于显示运行时的提示信息。这些界面资源提供了程序和用户之间的交互接口，在程序中合理布局这些界面元素，可以为用户提供良好的用户体验。本章将介绍这些界面资源的使用。

6.1　菜　　单

菜单是应用程序中常用的命令接口，为用户提供了触发事件的方式，并可以响应用户事件。菜单分为普通菜单和弹出式菜单。存放菜单的地方称为菜单栏。本节将介绍菜单及相关资源的使用。

6.1.1　菜单的种类及开发步骤

菜单就是将完成各种功能的菜单命令按照合理、易于使用和查找的方式布局在一起的菜单项集合，可分为普通菜单和弹出式菜单。普通菜单是停靠在菜单栏上的菜单。弹出式菜单是当用户右击某个区域时弹出的菜单，有时也称为快捷菜单。

在 Windows 系统中，菜单资源使用 HMENU 句柄标识。虽然可以使用 HMENU 句柄通过 Windows API 对菜单进行编程，但是开发过程很繁琐。因此 Visual Studio 2010 为菜单的开发提供了可视化的支持。开发菜单的步骤为：

（1）使用菜单编辑器，在其中增加、删除或修改菜单栏上的菜单以及菜单上的菜单项。

（2）使用菜单消息处理机制，完成对菜单命令消息和菜单更新消息的处理。菜单命令消息用于处理对菜单命令的响应，使用菜单更新消息可以根据程序条件改变更新菜单项的启用/禁用状态。

6.1.2　创建和编辑菜单

Visual Studio 2010 使用菜单编辑器创建和编辑菜单。在菜单编辑器中，可以直接在菜单栏中编辑菜单和菜单项，并且“所见即所得”，即菜单编辑器当前的样式就是程序运行时的样式。步骤如下：

（1）为工程增加菜单资源，并打开要编辑的菜单栏。

（2）在菜单栏上选择新项框，即空的矩形区域，或者使用右箭头按键和左箭头按键，将选择框移动到新项框中，如图 6-1 所示。

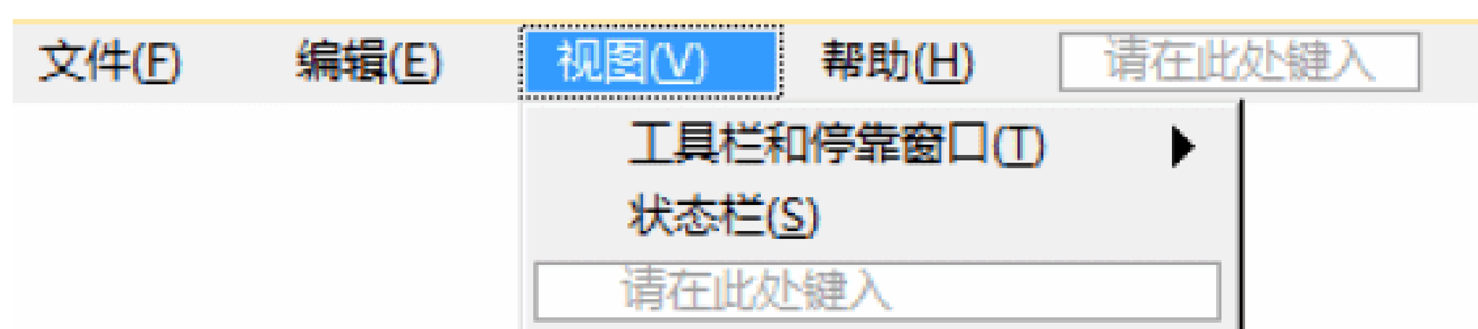


图 6-1　菜单编辑框

（3）输入菜单的名称。

创建好菜单后，就需要在菜单上创建执行命令功能的菜单项了。创建菜单项的步骤如下：

（1）打开创建好的菜单。

（2）选择菜单的新项框或选择已经存在的菜单项，按下 Insert 按键，就会在菜单项前增加新菜单项。

（3）输入菜单项的名称。

（4）在“属性”对话框中，选择要使用的菜单项样式。

（5）在“属性”对话框中的 Prompt 文本框中，输入要在状态栏中显示的提示字符串。

（6）按下 Enter 键，完成菜单项的添加。

6.1.3　处理菜单命令消息

使用菜单命令消息可以完成用户触发菜单命令时执行的操作。MFC 的消息映射机制为菜单命令消息的处理提供了非常简便的方法。菜单项、工具栏按钮和快捷键都是可以产生命令的用户接口对象，这些对象都具有资源 ID，通过共享资源 ID，可以将各种用户接口对象关联起来。例如，具有相同资源 ID 的菜单项、工具按钮和快捷键会映射到同一个处理函数。在第 5 章中介绍消息时，讲过命令消息是特殊的消息，因此，菜单命令消息的处理过程与消息的处理过程是一致的。如图 6-2 所示为菜单命令消息的处理机制。

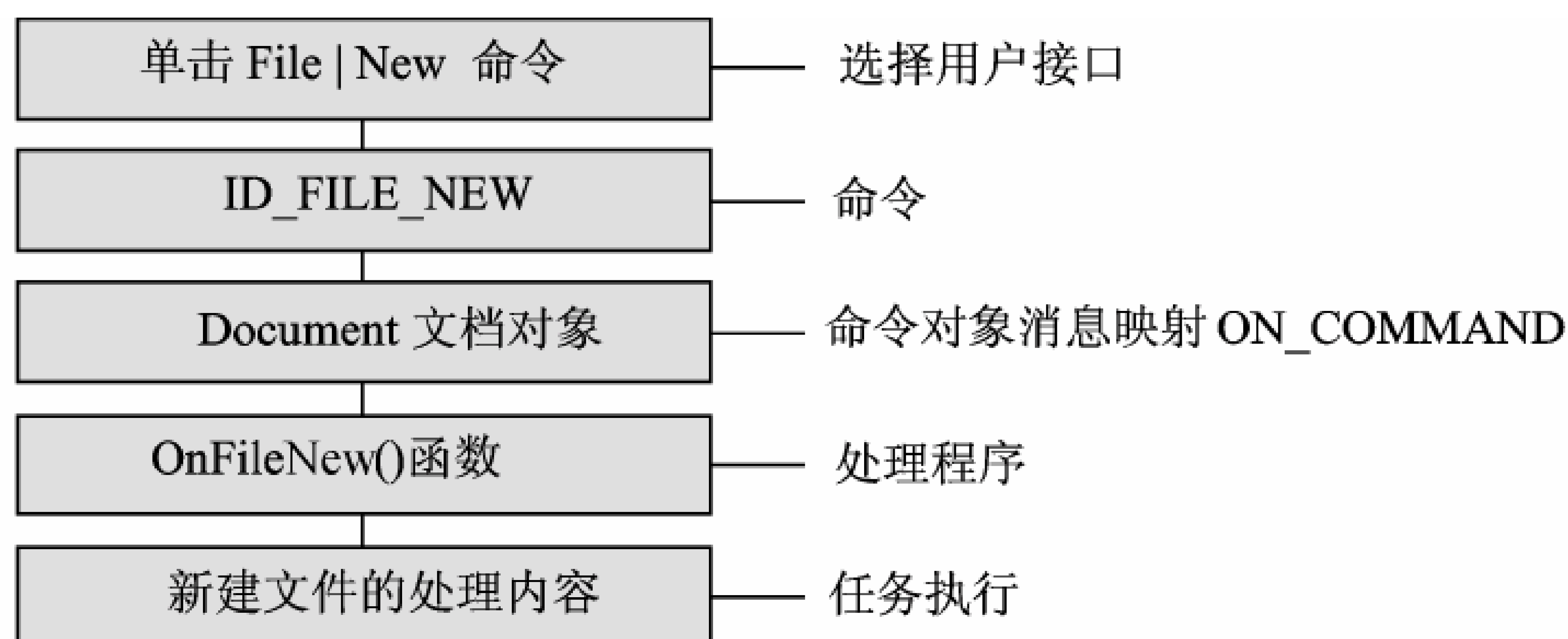


图 6-2　菜单命令消息处理流程图

图 6-2 显示了当用户单击 File|New 菜单项时，系统执行的操作。当用户单击菜单项时，系统判断消息 ID 为 ID_FILE_NEW，则根据 ON_COMMAND 的消息映射定位到 OnFileNew()函数，并执行其中的处理程序。在头文件中，菜单命令消息处理的实现代码如下：

```
01  class CMenuSampleDoc : public CDocument     //文档类声明
02  {
03  protected:
04      //{{AFX_MSG(CMenuSampleDoc)
```

```
05      afx_msg void OnMenuitemsettitle();     //声明设置标题命令处理函数
06      //}}AFX_MSG
07      DECLARE_MESSAGE_MAP()
08  };
```

在源文件中，代码如下：

```
01  BEGIN_MESSAGE_MAP(CMenuSampleDoc, CDocument)
02      //{{AFX_MSG_MAP(CMenuSampleDoc)
03      ON_COMMAND(ID_MENUITEMSETTITLE, OnMenuitemsettitle)    //消息映射
04      //}}AFX_MSG_MAP
05  END_MESSAGE_MAP()
06  //定义设置标题命令处理函数
07  void CMenuSampleDoc::OnMenuitemsettitle()
08  {
09      //设置文档标题
10      SetTitle(
11          CTime::GetCurrentTime().Format("log %Y-%m-%d %H:%M:%S")
12          );
13  }
```

上面代码显示了如何实现 ID_MENUITEMSETTITLE 菜单命令消息的处理。此消息的作用是根据当前时间设置文档标题，运行效果如图 6-3 所示。

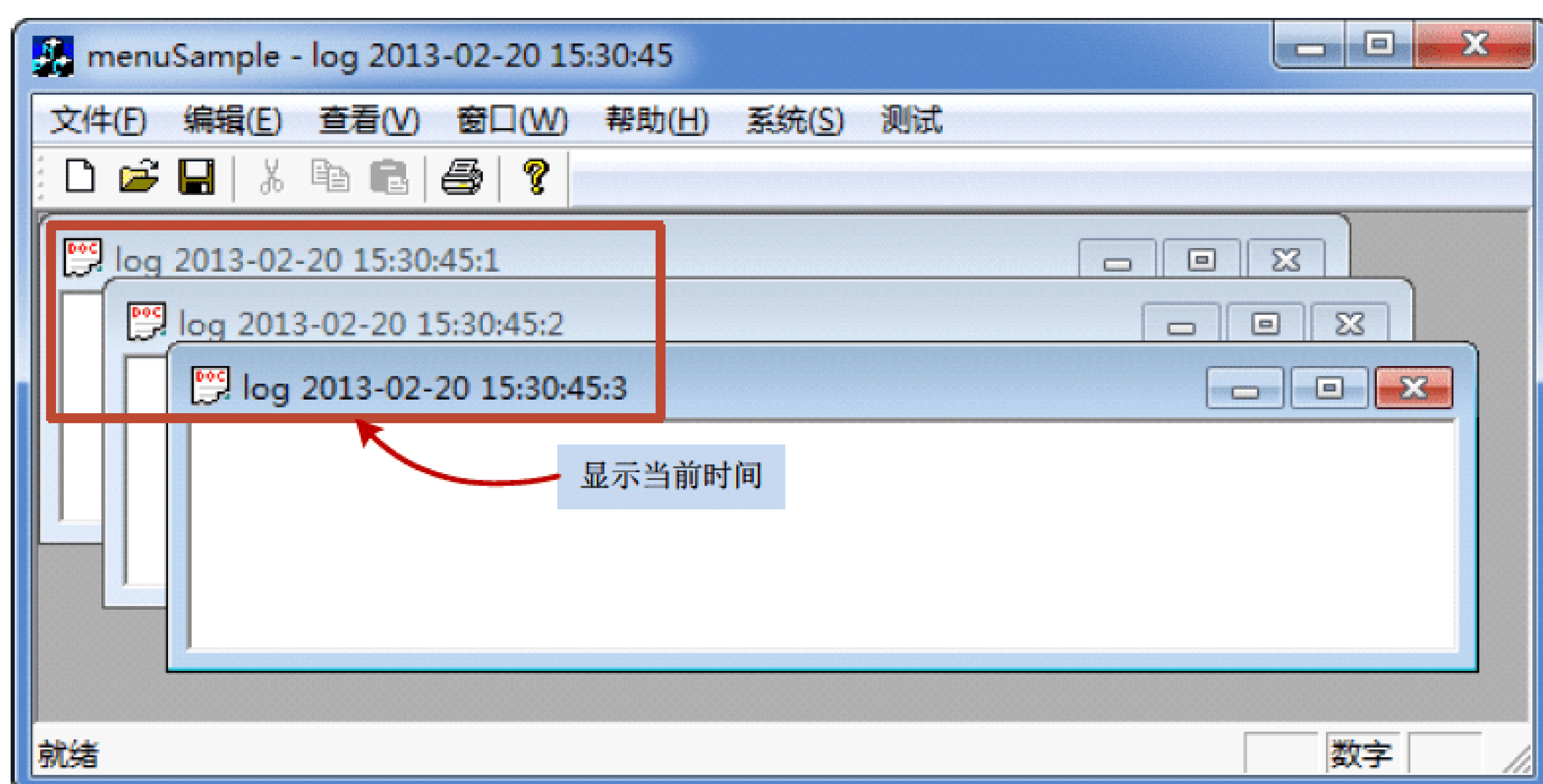


图 6-3　命令消息处理实例运行效果

6.1.4　处理菜单更新消息

在 Windows 程序中，菜单命令是有状态的，例如菜单是否选中，菜单是否可用等。使用菜单更新消息即可实现这些状态的控制。一般情况下，在程序的文档类中控制菜单更新状态是比较合理的。对于同一资源 ID 对应多个用户接口对象的情况，则处理一个菜单更新消息会影响所有的用户接口对象的状态。

当用户单击菜单时，系统会发送一个 WM_INITMENUPOPUP 消息，框架的更新机制会在菜单按下前同时向所有菜单项发送更新消息。如果存在菜单项的更新消息处理函数，则程序会调用此处理函数，如果更新消息函数不存在，则判断菜单消息处理函数是否存在并执行，最后根据情况启用或禁用相应的菜单项。如图 6-4 显示了菜单按下前 MFC 处理更新消息的过程。

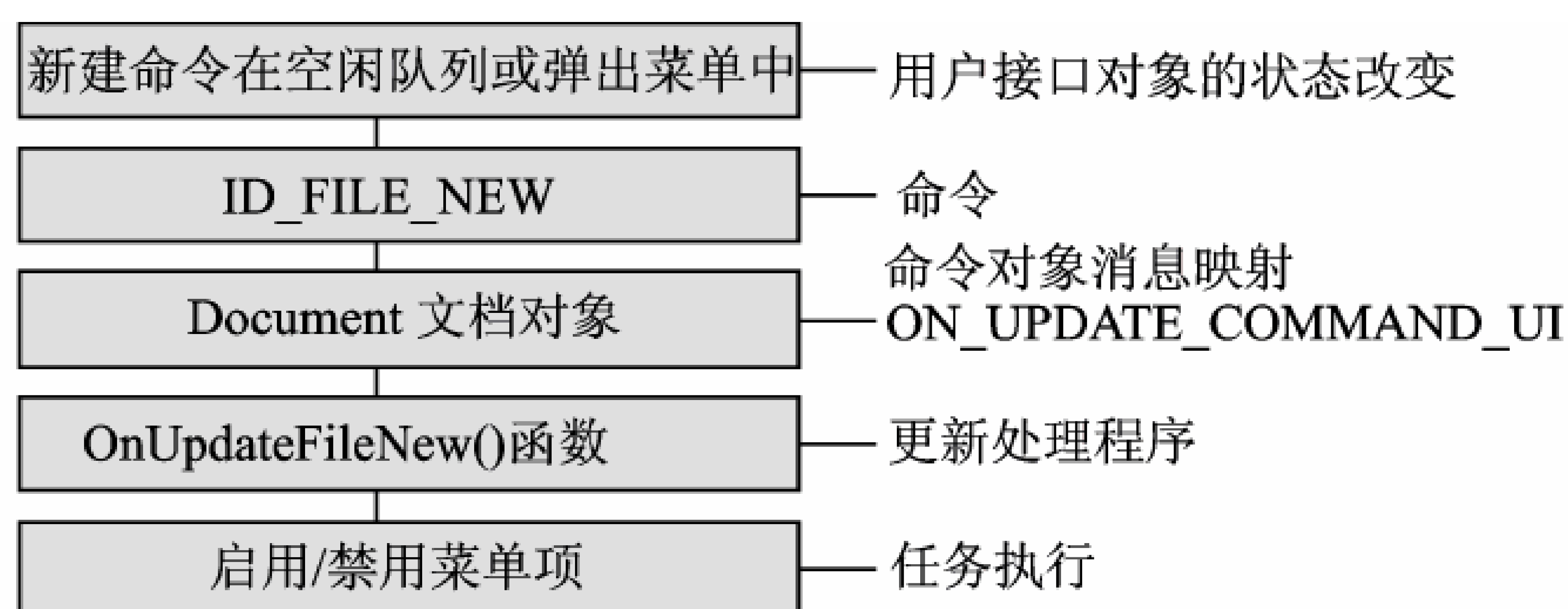


图 6-4　MFC 更新用户接口对象的过程

菜单更新消息的处理实现与菜单消息的处理实现是类似的。代码如下：

```
01  //头文件
02  class CMenuSampleView : public CView        //视图类声明
03  {
04  protected:
05      BOOL bToosCheck;                         //菜单项当前的选择状态
06      //{{AFX_MSG(CMenuSampleView)
07      afx_msg void OnMenuitemStatu();          //状态命令处理函数声明
08      //状态命令更新消息处理函数声明
09      afx_msg void OnUpdateMenuitemStatu(CCmdUI* pCmdUI);
10      //}}AFX_MSG
11      DECLARE_MESSAGE_MAP()
12  };
13  //源文件
14  BEGIN MESSAGE MAP(CMenuSampleView, CView)
15      //{{AFX_MSG_MAP(CMenuSampleView)         //消息映射
16      ON_COMMAND(ID_MENUITEM_STATU, OnMenuitemStatu)
17      ON_UPDATE_COMMAND_UI(ID_MENUITEM_STATU, OnUpdateMenuitemStatu)
18      //}}AFX_MSG_MAP
19  END_MESSAGE_MAP()
20  void CMenuSampleView::OnMenuitemStatu()     //状态测试命令消息处理函数
21  {
22      if (bToosCheck)
23          bToosCheck = FALSE;                  //切换菜单项的选择状态变量值
24      else
25          bToosCheck = TRUE;
26  }
27  void CMenuSampleView::OnUpdateMenuitemStatu(CCmdUI* pCmdUI)
28  {   //状态测试命令更新消息处理函数
29      if (bToosCheck)                          //判断菜单项状态变量
30      {
31          pCmdUI->SetCheck(1);     //如果选中，则设置菜单项状态为选中
32          pCmdUI->SetText("选择，单击取消");     //设置菜单项文本
33      }
34      else
35      {
36          pCmdUI->SetCheck(0);     //如果未选，则设置菜单项状态为未选
37          pCmdUI->SetText("未选择，单击选择");  //设置菜单项文本
38      }
39  }
```

上面的代码以一个示例，演示了如何实现菜单更新消息的实现。菜单更新消息处理函数的参数是 CCmdUI 类型的参数，表示发送更新消息的命令对象，此处是指 ID_MENUITEM_STATU 菜单项，因此使用 pCmdUI 变量可以设置菜单项的当前状态。程

序定义菜单项状态变量 bToosCheck 存储当前菜单项的状态，每次单击 ID_MENUITEM_STATU 菜单项，会根据当前状态值设置菜单项的样式并切换状态变量值，当下次单击菜单项时，会显示相反的状态。程序运行效果如图 6-5 所示。

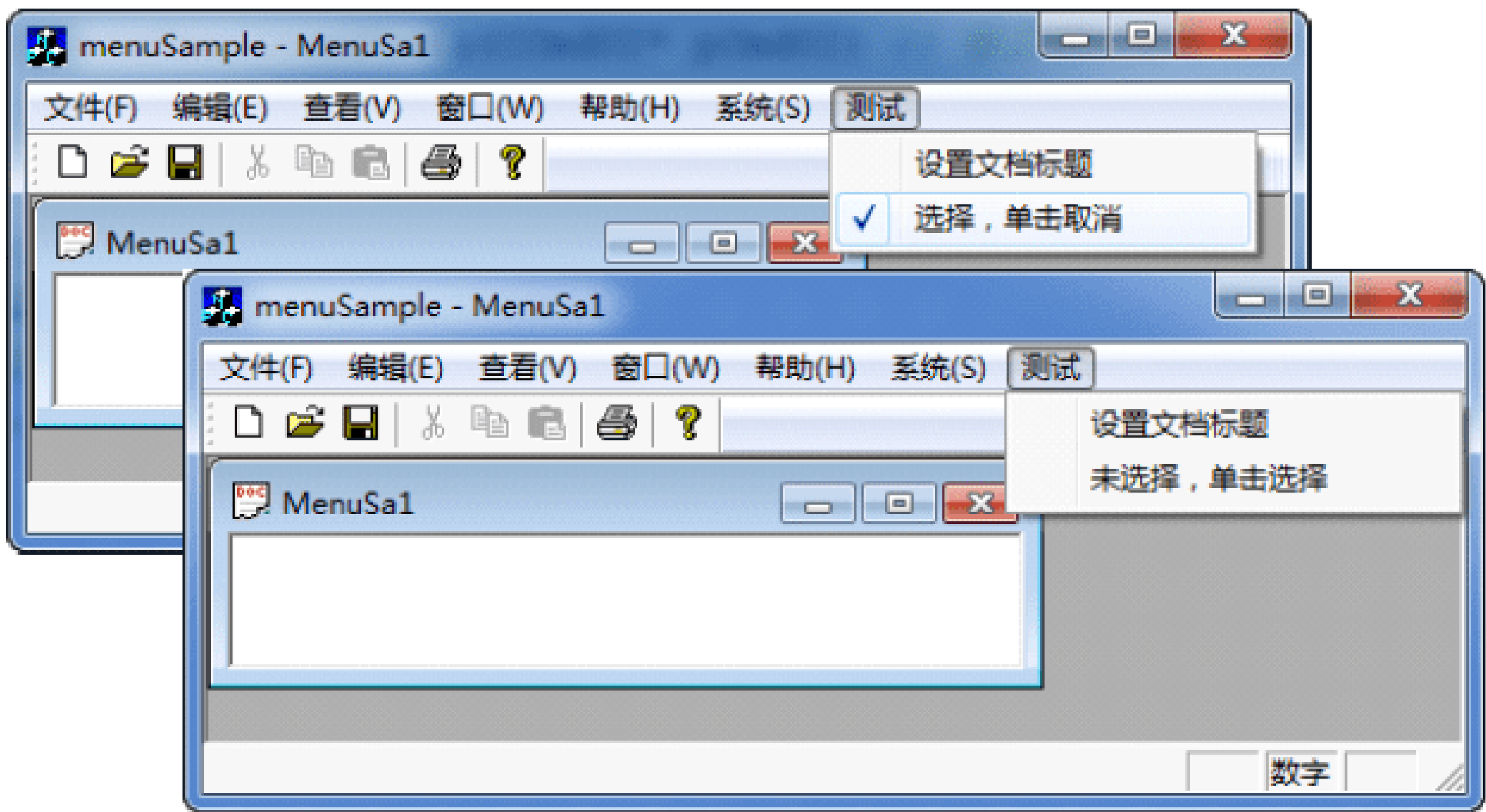


图 6-5　菜单更新消息示例运行效果

6.1.5　设置菜单项快捷键

在程序中，有时为了提高用户界面易用性，会为菜单项提供快捷键。快捷键与其对应的菜单项执行相同的命令消息处理函数和相同的命令更新消息处理函数。在 Visual Studio 2010 中，可以使用菜单编辑器为菜单项设置快捷键。具体步骤如下：

（1）在菜单编辑器中，选择要设置快捷键的菜单项。选择 View|Properties 命令或双击选择的菜单项，打开“属性”对话框。

（2）在 Caption 文本框中，输入菜单项名称+Tab 键转义字符（\t）+快捷键。例如要分配 File|Cut 命令的快捷键为 Ctrl+X 快捷键，则应该修改菜单项的文本为：

```
剪切(&T)\tCtrl+X
```

（3）在快捷键编辑器中创建一条快捷键条目，并分配标识符为菜单项标识符，如图 6-6 所示。

ID	修饰符	键	类型
ID_EDIT_COPY	Ctrl	C	VIRTKEY
ID_EDIT_COPY	Ctrl	VK_INSERT	VIRTKEY
ID_EDIT_CUT	Shift	VK_DELETE	VIRTKEY
ID_EDIT_CUT	Ctrl	X	VIRTKEY
ID_EDIT_PASTE	Ctrl	V	VIRTKEY
ID_EDIT_PASTE	Shift	VK_INSERT	VIRTKEY
ID_EDIT_UNDO	Alt	VK_BACK	VIRTKEY

图 6-6　菜单快捷键定义

6.1.6　创建与使用弹出式菜单

除了普通菜单，Visual Studio 2010 还提供弹出式菜单（也称为快捷菜单），当用户右击某一区域时，程序可以根据当前上下文环境弹出相应的弹出式菜单，并在其中显示常用命令。创建弹出式菜单分为两步——创建菜单并将其连接到上下文环境。当用户单击弹出

式菜单项时，会发送命令消息给窗口并调用命令消息处理函数。具体步骤如下：

（1）创建带有空标题的菜单栏。

（2）移动鼠标到菜单的第一个菜单项。打开属性页，输入标题和其他信息。根据需要依次创建需要的所有菜单项，保存当前菜单资源。

（3）在需要使用弹出式菜单的上下文环境对应的代码中，增加关联代码。代码如下：

```
01  //右击打开弹出式菜单
02  void CMenuSampleView::OnRButtonDown(UINT nFlags, CPoint point)
03  {
04      POINT screenPoint = point;      //定义屏幕坐标
05      ClientToScreen(&screenPoint);   //将传入的客户区域坐标转换为屏幕坐标
06      CMenu menu;                     //定义菜单对象
07      VERIFY(menu.LoadMenu(IDR_MENU_POPTEST));        //验证装载的菜单项
08      CMenu* pPopup = menu.GetSubMenu(0);             //获取弹出菜单
09      ASSERT(pPopup != NULL);                         //验证获取的弹出菜单
10      //显示弹出式菜单
11      pPopup->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
12          screenPoint.x, screenPoint.y, AfxGetMainWnd());
13      CView::OnRButtonDown(nFlags, point);        //调用基类的右键处理函数
14  }
```

（4）在对应的类中，使用前面介绍过的命令消息函数和命令更新消息函数的处理方式，为弹出菜单项添加消息处理代码。代码如下：

```
01  //弹出菜单 1 菜单项的处理函数
02  void CMenuSampleView::OnMenuitemPopItem1()
03  {
04      CDC* pDC = GetDC();                         //获取设备上下文
05      pDC->TextOut(0, 0, "单击弹出菜单中的菜单项 1");//在视图上显示提示信息
06  }
07  //弹出菜单 2 菜单项的处理函数
08  void CMenuSampleView::OnMenuitemPopItem2()
09  {
10      CDC* pDC = GetDC();                         //获取设备上下文
11      pDC->TextOut(0, 0, "单击弹出菜单中的菜单项 2");//在视图上显示提示信息
12  }
```

上面的代码为 CMenuSampleView 视图添加了弹出菜单，其中包含两个菜单项，分别是“弹出菜单项 1”和“弹出菜单项 2”，并为这两个菜单项增加了消息处理函数，消息处理函数会在当前视图显示触发的菜单项提示信息。运行效果如图 6-7 所示。

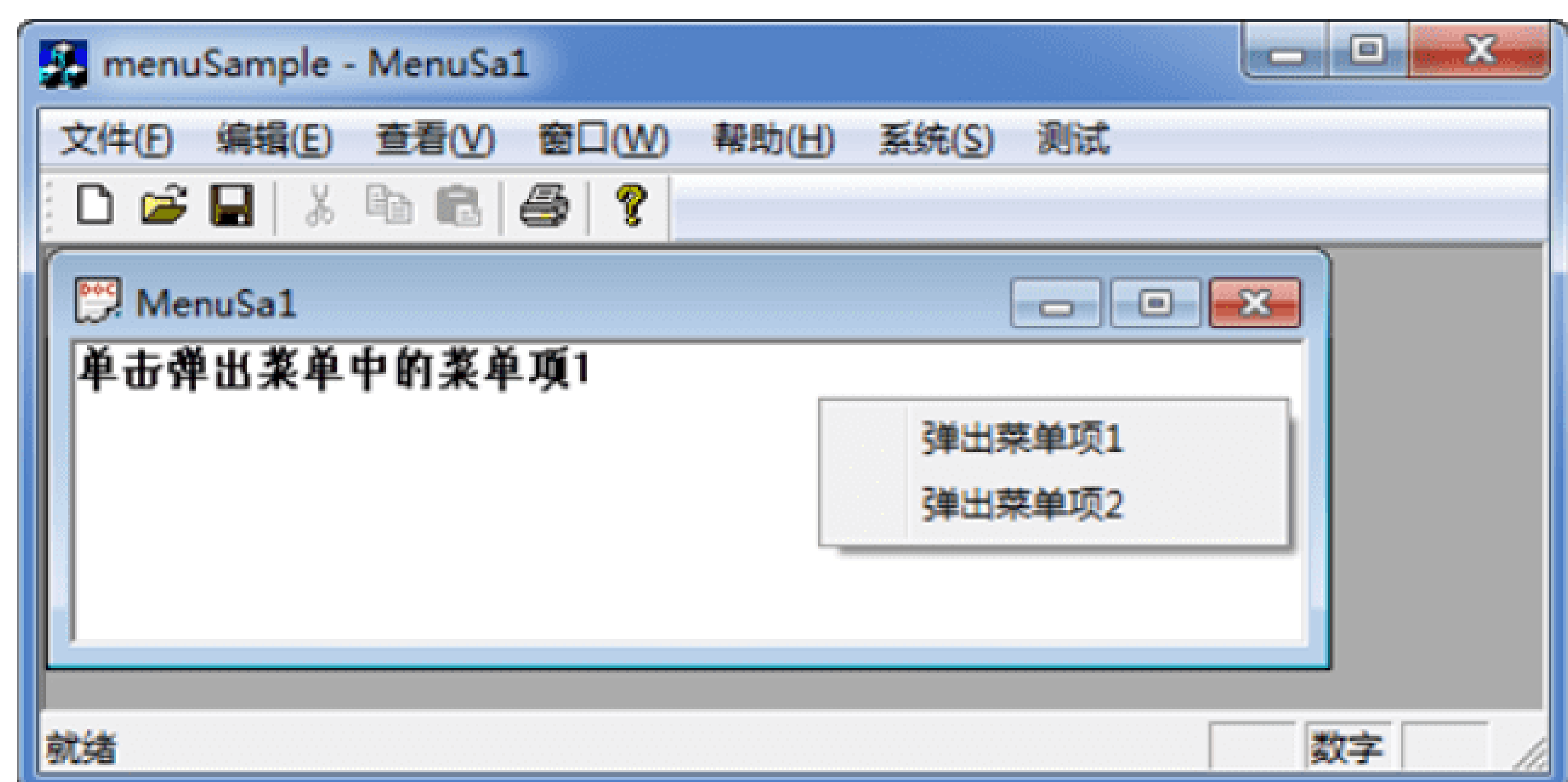


图 6-7　弹出菜单消息处理示例运行效果

6.1.7　菜单类 CMenu

CMenu 类封装了 Windows 的 HMENU 句柄，并提供了创建菜单、跟踪菜单、更新菜单和销毁菜单的成员函数。MFC 程序中，在框架类中定义和操作 CMenu 对象。代码如下：

```
01  //框架创建函数
02  int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
03  {
04      CMenu* menu = new CMenu();                     //定义菜单项
05      VERIFY(menu->LoadMenu(IDR_MENU1));             //验证装载的菜单项
06
07      if (!SetMenu(menu))                            //设置框架使用自定义菜单
08      {
09          TRACE0("创建菜单失败\n");                   //显示错误提示
10          return -1;                                 //创建失败，则返回
11      }
12  }
```

同时修改文档模板，代码如下：

```
01  pDocTemplate = new CMultiDocTemplate(
02      IDR_MENU1,
03      RUNTIME_CLASS(CMenuSampleDoc),
04      RUNTIME_CLASS(CChildFrame),
05      RUNTIME_CLASS(CMenuSampleView));
06  AddDocTemplate(pDocTemplate);
```

上面代码首先创建了 CMenu 对象指针，调用 CMenu 类的 LoadMenu()函数装载资源 ID 为 IDR_MENU1 的菜单。然后调用框架类的 SetMenu()函数设置框架对象使用用户自定义的菜单。程序运行效果如图 6-8 所示。

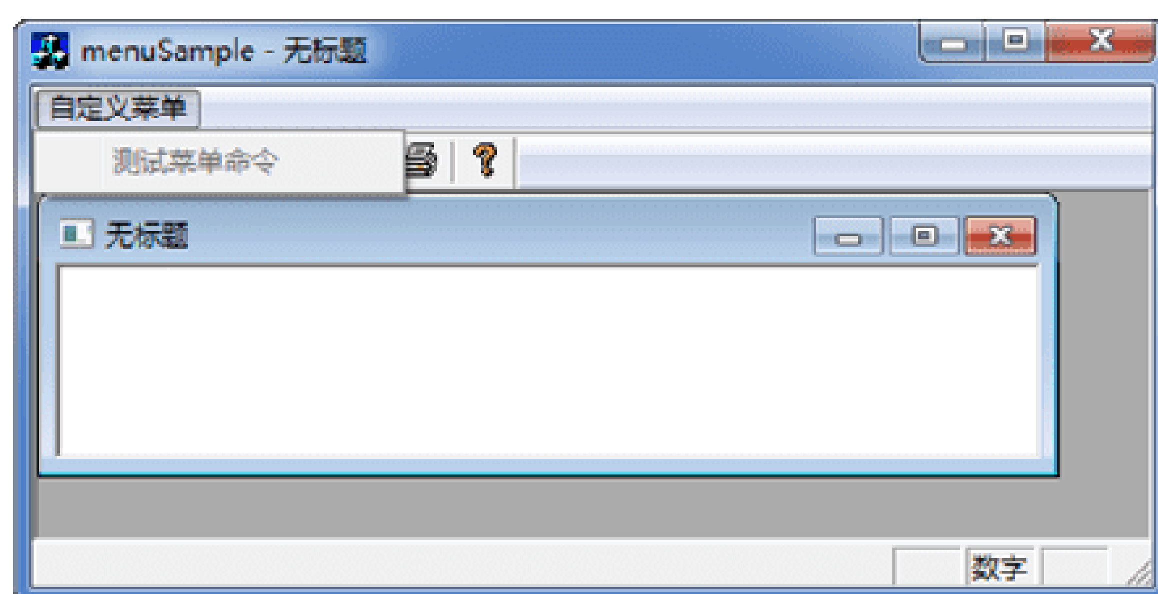


图 6-8　装载自定义菜单运行效果

从图 6-8 中可以看出，程序框架使用自定义菜单，没有使用系统默认的菜单。CMenu 类中只有一个数据成员，即 m_hMenu 成员，存储与 CMenu 对象绑定的 Windows 菜单句柄。当然，CMenu 类除了提供装载函数外，还提供了丰富的操作菜单及菜单项的成员函数，如表 6-1 所示。

表 6-1　CMenu的主要成员函数

成 员 函 数	功　　能
Attach()	绑定 Windows 菜单句柄到 CMenu 对象
Detach()	撤销 Windows 菜单句柄到 CMenu 对象的绑定，并返回菜单句柄
FromHandle()	返回菜单句柄对应的 CMenu 对象

续表

成 员 函 数	功　　能
GetSafeHmenu()	返回 CMenu 对象对应的 m_hMenu 菜单句柄
CreateMenu()	创建空菜单并将其绑定到 CMenu 对象
CreatePopupMenu()	创建空的弹出式菜单并将其绑定到 CMenu 对象
LoadMenu()	从可执行文件中装载菜单资源并将其绑定到 CMenu 对象
LoadMenuIndirect()	从内存中的菜单模板中装载菜单并将其绑定到 CMenu 对象
DestroyMenu()	销毁与 CMenu 绑定的菜单并释放菜单占用的内存
DeleteMenu()	删除菜单中指定的菜单项。如果删除的菜单项有与之关联的弹出菜单，则会销毁弹出式菜单并释放使用的内存
TrackPopupMenu()	在指定位置显示浮动的弹出式菜单，并跟踪弹出菜单的选择项
AppendMenu()	向菜单尾添加新菜单项
CheckMenuItem()	在弹出菜单中的菜单项上标记选择标志或移除选择标志
CheckMenuRadioItem()	选择指定菜单项，并将其分组中的其他菜单项设置为未选择
SetDefaultItem()	为指定菜单设置默认的菜单项
GetDefaultItem()	获取指定菜单的默认菜单项
EnableMenuItem()	启用、关闭和变灰菜单项
GetMenuItemCount()	获取菜单中包含的菜单项，只计算弹出菜单或顶层菜单的菜单项数目
GetMenuItemID()	获取指定位置菜单项的菜单项标识符
GetMenuState()	获取指定菜单项的状态或弹出菜单的菜单项数目
GetMenuString()	获取指定菜单项的文本
GetMenuItemInfo()	获取菜单项信息
GetSubMenu()	返回弹出菜单的指针
InsertMenu()	在指定位置插入新菜单项
ModifyMenu()	修改指定位置的菜单项
RemoveMenu()	删除菜单上的菜单项以及与此菜单项相连的弹出菜单
SetMenuItemBitmaps()	设置菜单项的选择图像
GetMenuContextHelpId()	返回与菜单相连的帮助上下文 ID
SetMenuContextHelpId()	设置与菜单相连的帮助上下文 ID

6.2　工　具　栏

除了使用菜单可以触发事件外，使用工具栏也可以触发命令。工具栏是菜单的补充，是菜单的快捷方式，是存放一组相关命令按钮的控件。一般情况下，工具栏中放置的命令按钮与常用的菜单项之间是关联的。本节将主要介绍工具栏的使用。

6.2.1　创建与编辑工具栏

工具栏中存放一组相关的命令按钮。工具栏上的命令按钮与菜单中的菜单项的工作原理是一致的，两者都是用户接口对象，只需要将工具栏按钮的资源 ID 赋值为对应的菜单项的资源 ID，就可以使工具栏按钮和菜单项相连。工具栏命令按钮像菜单项一样是具有状态的，而且可以将其实现为按钮、单选按钮或复选框。

Visual Studio 2010 提供了工具栏编辑器创建和编辑工具栏。在工具栏编辑器中，可以

直接在工具栏中编辑工具栏和命令按钮，并且是所见即所得，即工具栏编辑器当前的样式就是程序运行时的样式。步骤如下：

（1）为工程增加工具栏资源，并打开要编辑的工具栏。

（2）在工具栏上单击新项框，即空的矩形区域，或者使用右箭头按键和左箭头按键将选择框移动到新项框中，双击新项框，打开“属性”对话框，如图 6-9 所示。

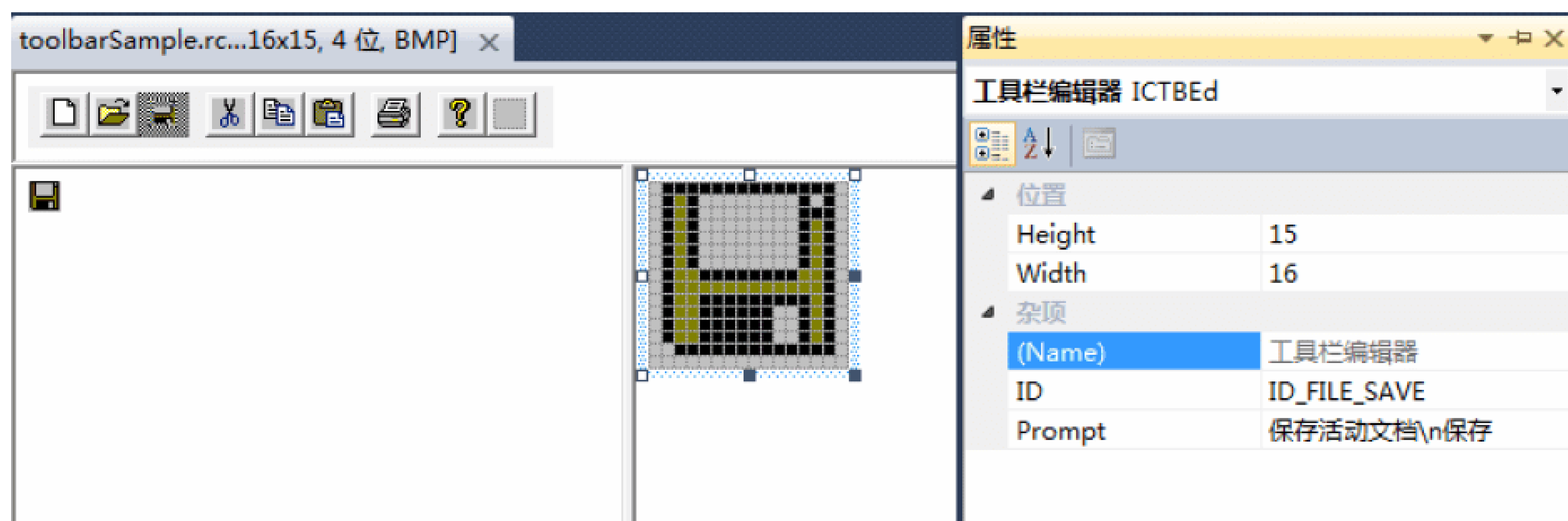


图 6-9　工具栏编辑“属性”对话框

（3）在 ID 文本框中输入命令按钮对应的菜单项的资源 ID。注意，命令按钮是没有样式设置的，因为它是菜单项的快捷方式，因此，要设置命令按钮的样式，需要设置对应的菜单项的样式。如图 6-9 中，是“保存”菜单项对应的命令按钮的设置，命令按钮的 ID 值为 ID_FILE_SAVE，表示对应“保存”菜单项。

（4）在程序编写过程中，可以根据实际情况，使用上面的步骤修改命令按钮的图片、大小和对应的菜单项的资源 ID。

6.2.2　设置工具栏停靠和浮动

工具栏与菜单最主要的一个区别是工具栏是可以停靠在指定窗口或浮动的，而菜单的位置是固定的。要控制工具栏的停靠和浮动，需要完成下面的工作。

（1）调用 CFrameWnd::EnableDocking()函数设置可以在框架窗口中停靠界面对象。此函数的参数是 DWORD 类型的，表示框架窗口的哪几个边允许停靠界面对象。可以指定上边、下边、左边和右边或者是任何地方都可以停靠。

（2）调用 CControlBar::EnableDocking()函数设置工具栏可以停靠。参数也是 DWORD 类型，用于表示工具栏可以停靠的边。如果在 CControlBar::EnableDocking()调用中没有指定与框架对话框中可以停靠的边匹配的边，则工具栏不会停靠在窗口中，而是浮动在窗口上。如果要设置工具栏不能停靠在窗口边，则设置 EnableDocking()函数的参数为 0，并调用 CFrameWnd::FloatControlBar()函数即可。

（3）调用 CFrameWnd::DockControlBar()函数，可以使工具栏停靠到框架对话框。调用 CFrameWnd::FloatControlBar()函数，可以使工具栏浮动在窗口上。

（4）如果使用浮动工具栏，还可以设置工具栏的显示样式，比如设置显示样式是水平显示还是垂直显示，工具栏是否可以调整大小等。

下面的示例演示了停靠工具栏的使用过程。

```
01  //创建工具栏
02  if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
```

```
03          WS_VISIBLE | CBRS_TOP   | CBRS_GRIPPER | CBRS_TOOLTIPS |
04          CBRS_FLYBY | CBRS_SIZE_FIXED) ||
05          !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
06  {
07      TRACE0("Failed to create toolbar\n");  //输出错误信息，创建工具栏失败
08      return -1;                             //创建失败，则返回
09  }
10  //设置工具栏可以停靠在窗口的任意边
11  m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
12  EnableDocking(CBRS_ALIGN_ANY);          //框架设置允许界面对象停靠
13  DockControlBar(&m_wndToolBar);          //工具栏停靠
14  //获取第 5 个按钮的样式
15  UINT nStyle = m_wndToolBar.GetButtonStyle( 5 );
16  nStyle |= TBBS_WRAPPED;                 //设置按钮的样式为分隔符
17  m_wndToolBar.SetButtonStyle( 5, nStyle );//设置第 5 个按钮为分隔符
```

在 MFC 中工具栏对象声明为 CMainFrame 类的数据成员，也就是说，工具栏对象是内置在主框架对话框中的对象。当创建框架对话框时，MFC 创建工具栏。当销毁框架对话框时，销毁工具栏。上面代码中，创建了可以停靠的固定大小的工具栏，将工具栏停靠在窗口中，并设置工具栏上第 5 个按钮为分隔符，程序运行效果如图 6-10 所示。

6.2.3　设置工具提示

Visual Studio 2010 为界面的友好性也提供了基本的支持。其中工具提示是常见工具帮助的一种，当用户鼠标划过工具栏按钮一段时间后，系统会在状态栏中或弹出的小窗口中显示工具栏按钮的功能描述文本，其为用户提供的工具按钮使用说明。图 6-11 中显示了两种工具提示的样式。

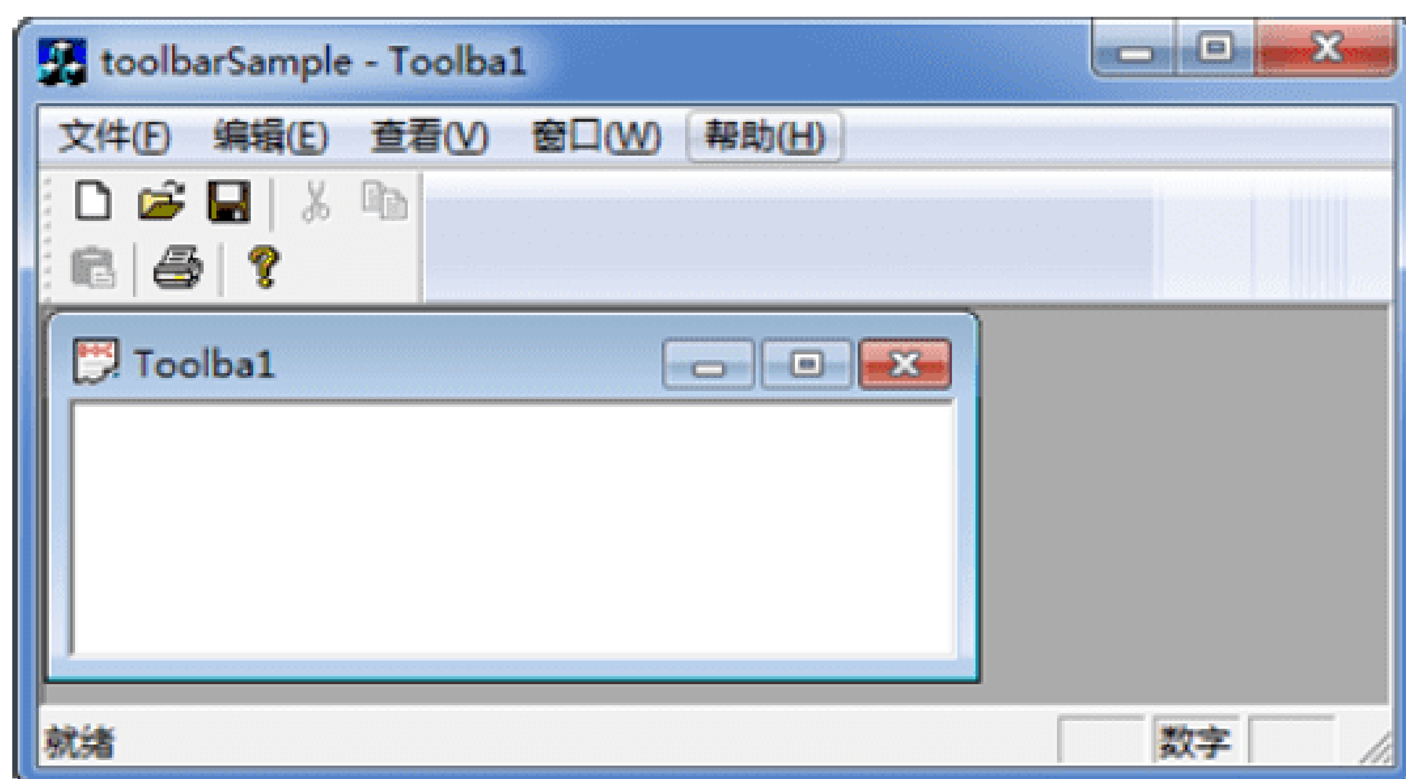


图 6-10　停靠工具栏示例运行效果

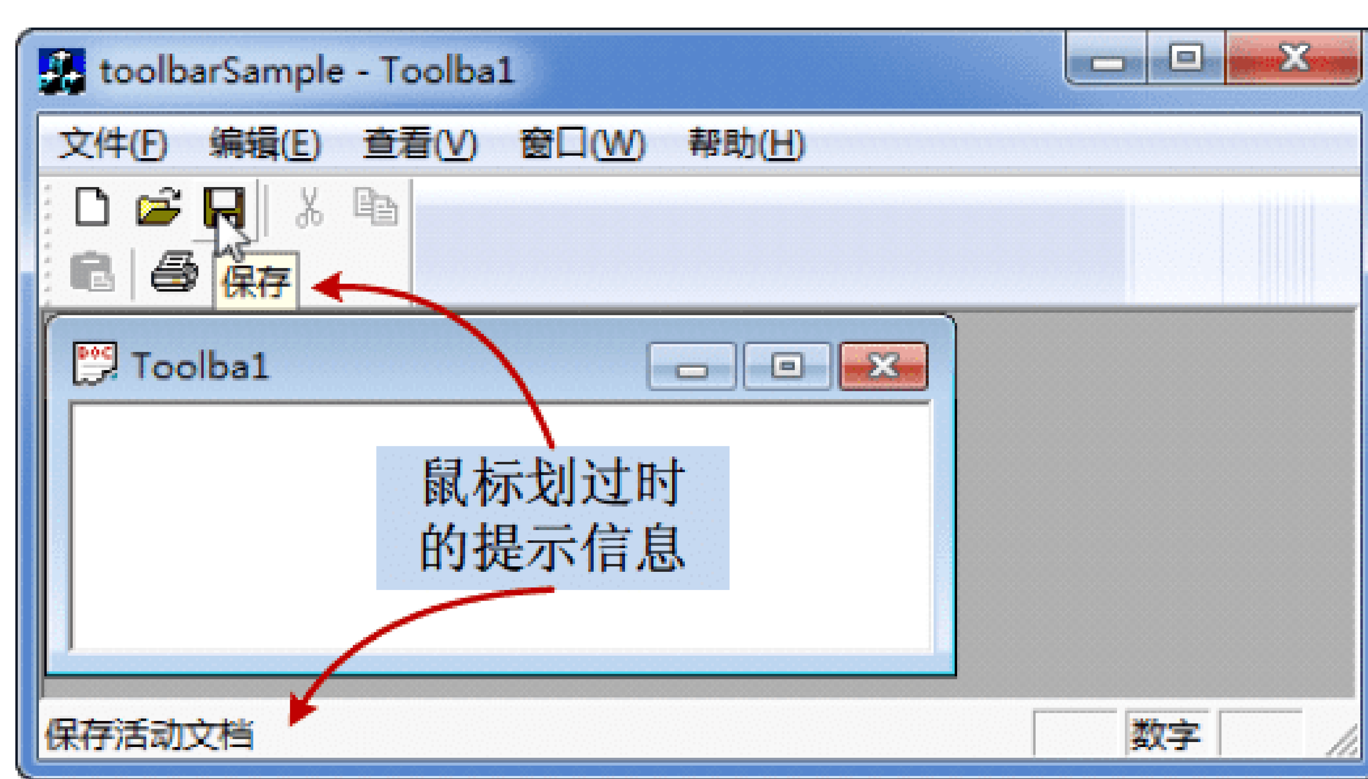


图 6-11　工具提示的样式

默认情况下，使用 Visual Studio 2010 应用向导创建的应用程序的工具栏是支持工具提示的。要在应用程序中支持工具提示功能，则需要：

- 设置工具栏的样式支持工具提示。在创建工具栏时，在样式中增加 CBRS_TOOLTIPS 样式。或在 SetBarStyle()函数中设置 CBRS_TOOLTIPS 样式。
- 添加工具提示文本。在字符串资源中创建与工具栏按钮的资源 ID 相同的字符串资源 ID，并将其值赋值为工具提示的文本。或者直接在工具栏编辑器中的“属性”对话框中的 Prompt 文本框中输入提示文本。
- 默认情况下，工具栏上的工具提示只有当工具被激活时，才会显示。如果要实现

当鼠标划过工具按钮时显示工具提示信息，则需要为工具栏的样式增加 CBRS_FLYBY 属性。

6.2.4　CToolBar 介绍

CToolBar 类用于表示工具栏，封装了 Windows 的工具栏句柄，包含一组位图按钮和分隔符。按钮可以是命令按钮、复选框和单选按钮。它内置了派生自 CFrameWnd 类的窗口对象的成员函数。MFC 程序中，在框架类中定义和操作 CToolBar 对象。具体代码如 6.2.2 小节中的示例。表 6-2 中列出了 CToolBar 类的成员函数。

表 6-2　CToolBar类的成员函数

函 数 名 称	功　　能
Create 或 CreateEx()	创建 Windows 工具栏，并将其附加到 CToolBar 对象
SetSizes()	设置按钮和图片的尺寸大小
SetHeight()	设置工具栏的高度
LoadToolBar()	装载资源编辑器创建的工具栏资源
LoadBitmap()	装载包含按钮位图的图像
SetBitmap()	设置位图图像
SetButtons()	设置按钮样式和图片在图像中的索引
CommandToIndex()	返回指定命令 ID 的命令索引
GetItemID()	获取指定位置的按钮或分隔符的命令 ID
GetItemRect()	返回给定索引处的按钮显示的矩形区域
GetButtonStyle()	返回按钮的样式
SetButtonStyle()	设置按钮样式
GetButtonInfo()	返回命令 ID、命令样式和按钮的图像索引
SetButtonInfo()	设置命令 ID、命令样式和按钮的图像索引
GetButtonText()	获取按钮上显示的文本
SetButtonText()	设置按钮上显示的文本
GetToolBarCtrl()	获取底层工具栏对象

6.3　状　态　栏

状态栏是显示在界面底部的具有一个或多个面板的控件栏，面板中包含文本或位图。使用面板可以显示程序运行时的各种状态信息，如命令描述、时间、页码或按键状态等。本节将介绍有关状态栏的使用。

6.3.1　创建状态栏

状态栏为应用程序提供了一种不中断用户工作而显示提示信息的方式。通常状态栏显示在对话框的底部，状态栏有“面板”，包括“指示器”和“消息行”。指示器显示 SCROLL LOCK 按键是否按下、宏记录是否打开等状态信息。消息行显示有关程序运行的信息。在

Visual Studio 2010 中没有提供可视的资源编辑器编辑状态栏。因此，要创建状态栏必须使用代码控制。创建状态栏的步骤为：

（1）在框架类中定义 CStatusBar 对象。

（2）在框架类的 OnCreate 中创建状态栏对象。代码如下：

```
01  //创建状态栏对象，并设置面板信息
02  if (!m_wndStatusBar.Create(this) ||
03              !m_wndStatusBar.SetIndicators(indicators,
04                              sizeof(indicators)/sizeof(UINT)))
05  {
06      //如果创建状态栏失败，则显示错误信息
07      TRACE0("Failed to create status bar\n");
08      return -1;                          //如果创建状态栏失败，则返回
09  }
```

（3）根据实际需求，设置状态栏面板的样式。状态栏是包含多个面板的 Window 对象，每个面板是状态栏上的一个矩形区域，可以显示信息。如很多应用程序在右边的面板上依次显示 CAPS LOCK、NUM LOCK 和其他键的状态，在左边面板上显示信息文本，如默认的 MFC 状态栏在右边显示当前选择的菜单项或工具栏按钮的功能描述字符串。读者可以根据自己的需要，修改状态栏面板中的文本。如图 6-12 所示，显示了使用 Visual Studio 2010 创建的程序的默认状态栏的样式。

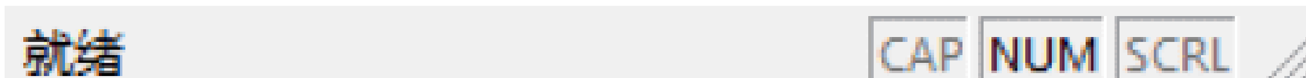


图 6-12　状态栏示例

（4）添加状态栏处理代码。因为面板与菜单项或命令按钮不同，不能发送命令消息 WM_COMMAND，但是通过 ON_UPDATE_COMMAND_UI 消息，可以实现与面板之间的互动。不过因为状态栏面板不是命令按钮，所以必须手动添加处理代码。

6.3.2　状态栏实例

下面以一个实例演示如何使用状态栏。此实例在状态栏的右边实时地显示当前时间。操作过程如下所示。

（1）右击资源视图的任意文件夹，在弹出的快捷菜单中选择“资源符号”命令，打开“资源符号”对话框，增加符号资源，如图 6-13 所示。

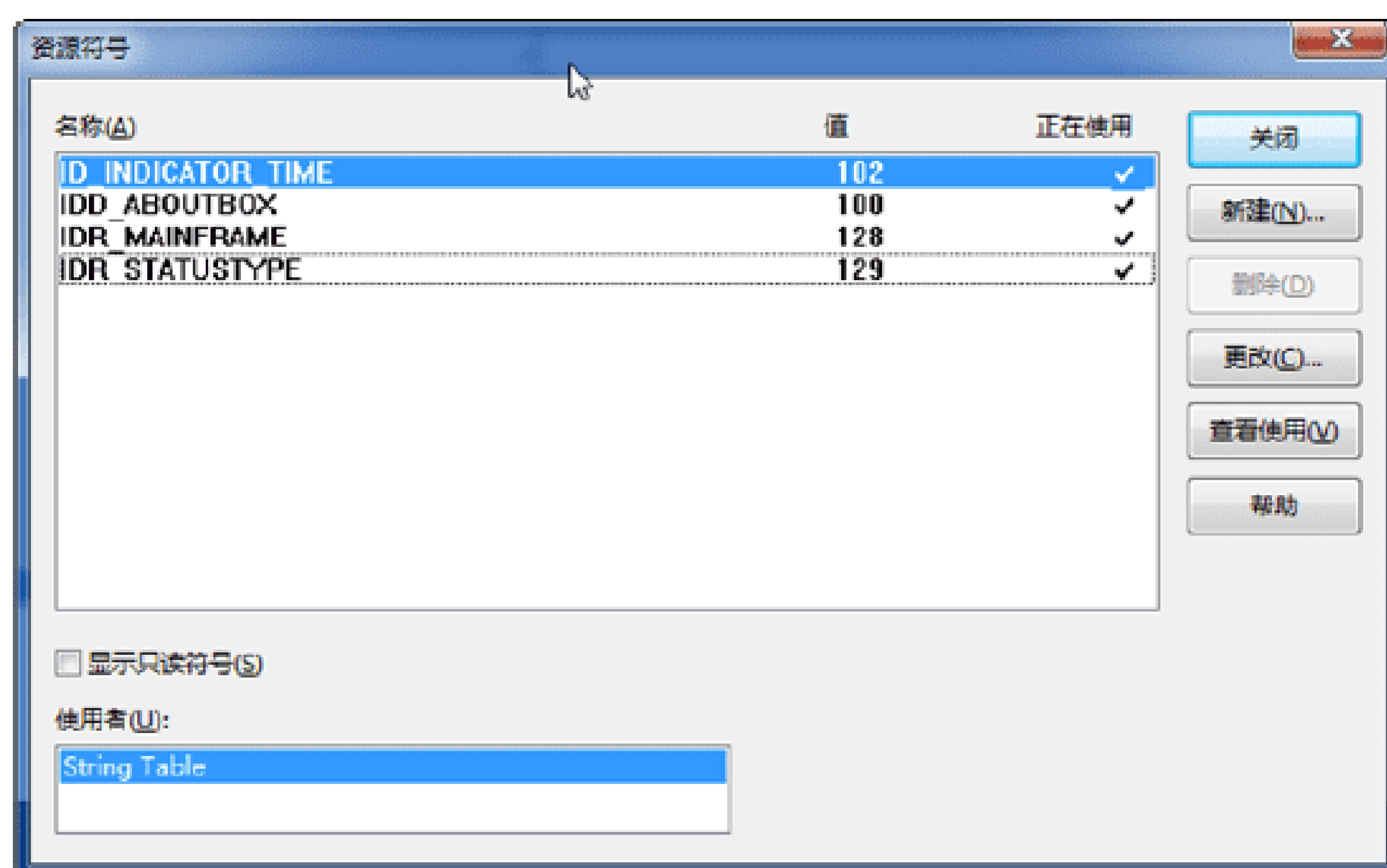


图 6-13　“资源符号”对话框

（2）单击“新建”按钮，打开“新建符号”对话框，在其中输入符号名称和取值，这里增加显示时间的面板符号资源，如图 6-14 所示。单击“确定”按钮，返回“资源符号”对话框。单击“关闭”按钮，关闭“资源符号”对话框。

（3）添加字符串资源，在其中选择面板对应的资源 ID，并输入时间面板的默认显示文本，如图 6-15 所示。

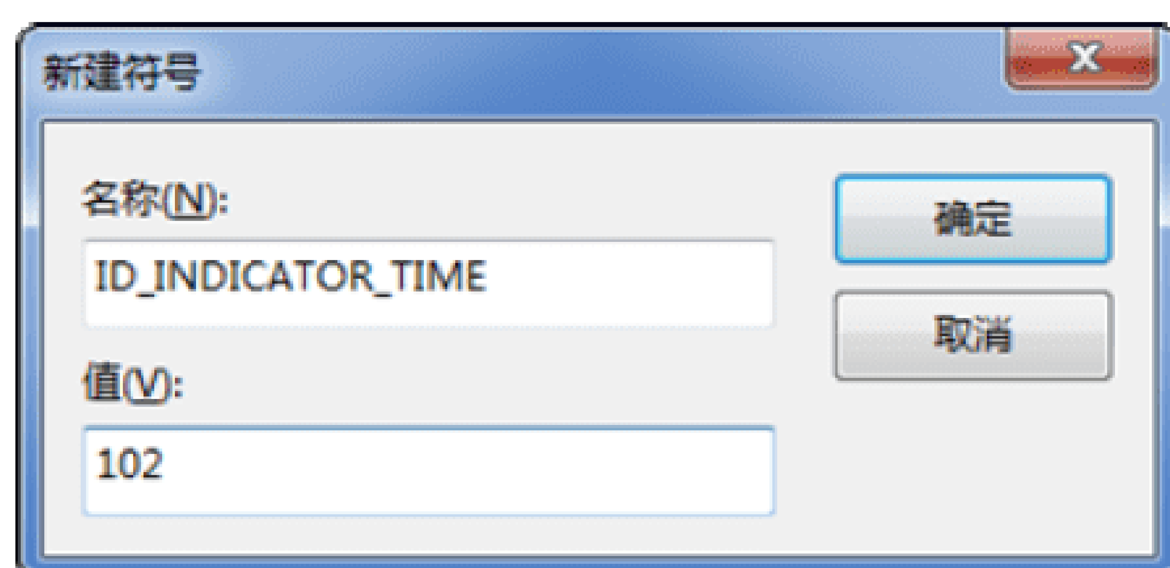


图 6-14 “新建符号”对话框

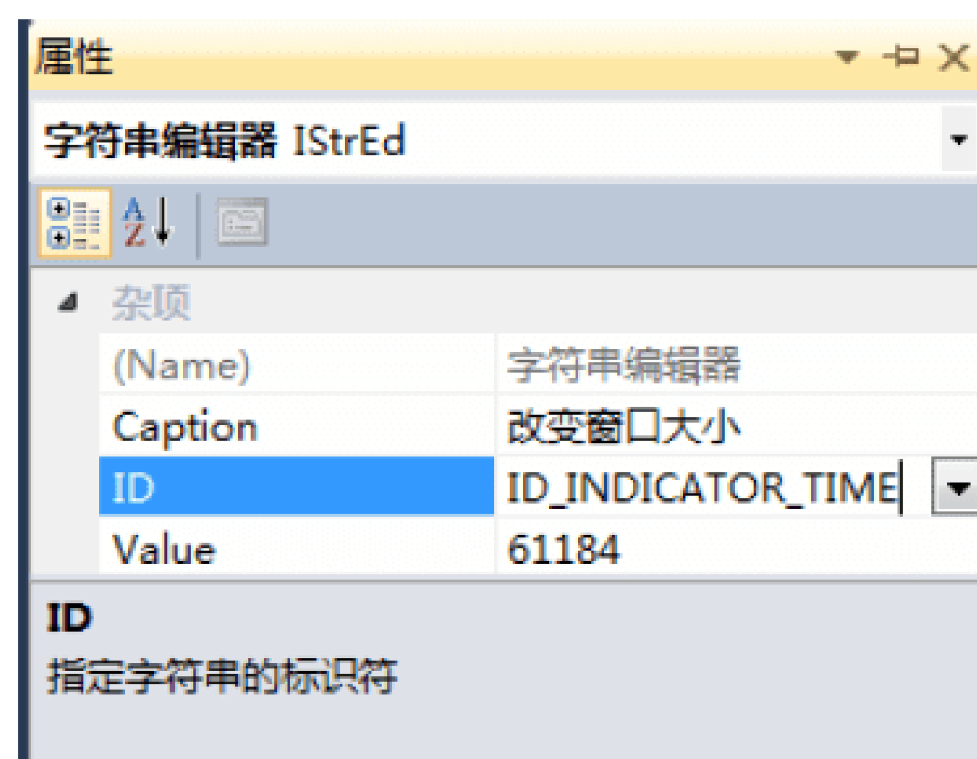


图 6-15 “属性”对话框

（4）将自定义面板增加到 indicators 数组中。indicators 数组从左到右定义了面板显示的内容。在要插入面板的位置点，输入增加的面板命令 ID，即 ID_INDICATOR_TIME。代码如下：

```
01  static UINT indicators[] =
02  {
03      ID_SEPARATOR,           //状态栏分组
04      ID_INDICATOR_CAPS,      //大小写指示
05      ID_INDICATOR_NUM,       //数字键指示
06      ID_INDICATOR_SCRL,      //滚动键指示
07      ID_INDICATOR_TIME,      //时间指示
08  };
```

如果增加的面板是显示固定信息，如公司名称等，则状态栏就增加成功了。对于有些动态信息，则面板增加到状态栏后，还需要动态地修改显示信息。但是，通常设置面板显示文本时在更新处理函数中调用 CCmdUI 对象的 SetText()成员函数。例如，在本例中要在时间面板上实时显示当前时间，则需要执行如下步骤，并使用 SetText()设置面板的文本来显示当前的时间。

（1）在框架对象中定义日期类型的变量 m_time 存储当前的时间。

（2）在框架类的声明中为时间面板增加更新处理函数声明。代码如下：

```
afx_msg void OnUpdateTime(CCmdUI *pCmdUI);      //更新消息函数声明
```

上面代码中增加的是时间面板的 OnUpdateTime()更新处理函数的声明。

（3）在框架类的源文件中为时间面板增加更新处理函数定义。代码如下：

```
01  void CMainFrame:: OnUpdateTime(CCmdUI *pCmdUI) //更新消息函数定义
02  {
03      pCmdUI->Enable();                         //使时间面板可用
04      //设置时间面板显示的文本为当前的时间
05      pCmdUI->SetText( m_time.Format( "%H:%M:%S" ) );
06  }
```

（4）在框架类的源文件中增加时间面板 ID_INDICATOR_TIME 的更新处理宏 ON_UPDATE_COMMAND_UI 与更新处理函数之间的消息映射。代码如下：

```
01  BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
02      //{{AFX_MSG_MAP(CMainFrame)
03      ON_WM_CREATE()                          //创建消息映射
04      //}}AFX_MSG_MAP
05      //TIME 更新消息映射
06      ON_UPDATE_COMMAND_UI(ID_INDICATOR_TIME,OnUpdateTime)
07  END_MESSAGE_MAP()
```

上面代码增加的时间面板的更新处理消息映射应该放在//{{AFX 注释段外，但是要在 BEGIN_MESSAGE_MAP 和 END_MESSAGE_MAP()之间的代码段中，这个代码段中定义所有的消息映射，//{{AFX_MSG_MAP 宏之间定义系统自定义消息映射。

（5）根据需要处理 m_time 变量的取值，本例中使用定时器每隔 1 秒更新 m_time 变量的值为当前时间。当进程空闲时，框架会将当前的 m_time 的值显示在时间面板上。代码如下：

```
01  void CMainFrame::OnTimer(UINT nIDEvent)     //定时器处理函数
02  {
03      //定时获取当前时间值
04      if (nIDEvent == 20)
05          m_time = CTime :: GetCurrentTime();
06      CMDIFrameWnd::OnTimer(nIDEvent);        //调用默认的定时器处理函数
07  }
08  ...
09  //在 OnCreate()函数中启动 ID 为 20、时间间隔为 1 秒的定时器
10  SetTimer(20, 1000, NULL);
11  ...
12  //在析构函数中停止 ID 为 20 的定时器
13  KillTimer(20);
```

这样，定时显示当前时间的状态栏面板就完成了。程序运行效果如图 6-16 所示。

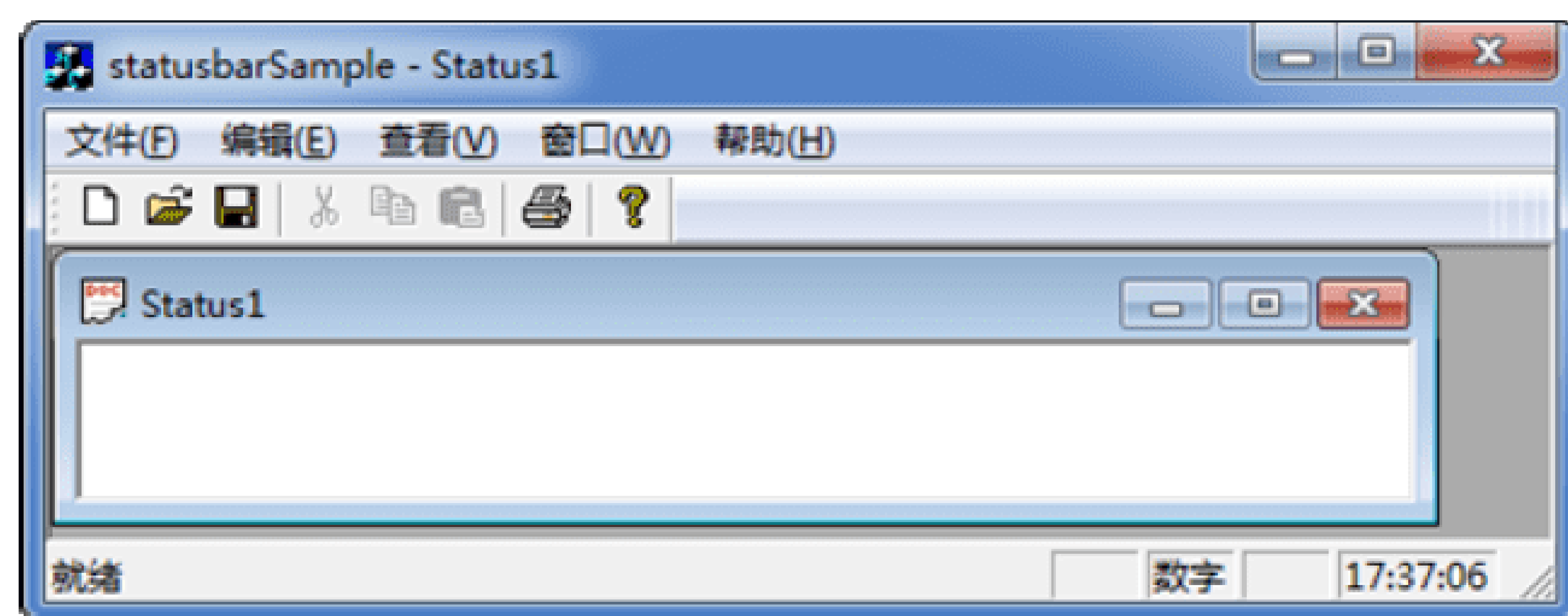


图 6-16　显示时间的状态栏实例运行效果

6.3.3　CStatusBar 介绍

MFC 中使用 CStatusBar 类封装状态栏对象 CStatusBarCtrl。使用 CStatusBar 对象的成员函数可以设置状态栏的多种特性。与工具栏一样，状态栏对象内置在框架对话框中。当框架对话框构造时，自动构造状态栏，并在析构函数中自动析构。表 6-3 中列出了 CStatusBar 类的常用成员函数。

表 6-3　CStatusBar类的常用成员函数

成 员 函 数	构造 CStatusBar 对象
Create 或 CreateEx()	创建工具栏，将其附加到 CStatusBar 对象中，设置初始字体和工具栏高度
SetIndicators()	设置指示符 ID
CommandToIndex()	获取指定指示符 ID 的索引
GetItemID()	获取指定索引处的指示符 ID
GetItemRect()	获取指定索引处的显示矩形区域
GetPaneInfo()	获取指定索引处的面板信息，包括指示符 ID、样式和宽度
GetPaneStyle()	获取指定索引处的面板的样式
GetPaneText()	获取指定索引处的面板的文本
GetStatusBarCtrl()	返回工具栏对象对应的底层控件
SetPaneStyle()	设置指定索引处的面板样式
SetPaneText()	设置指定索引处的面板文本
SetPaneInfo()	设置指定索引处的面板信息，包括指示符 ID、样式和宽度

6.4　本 章 小 结

本章主要介绍了程序中常用的用户界面对象菜单、工具栏和状态栏。本章的重点是在 MFC 程序中使用菜单、工具栏和状态栏的方法。本章的难点是如何使用 CMenu、CToolBar 和 CStatusBar 操作菜单、工具栏和状态栏。第 7 章将介绍窗口中的界面元素——Windows 标准控件的使用。

6.5　习　　题

1．为 MFC 单文档应用程序添加一个菜单项“自定义菜单”，它有一个子菜单“修改窗体标题”，当子菜单项被单击时会修改主窗体的名称为“习题 6.5 第 1 题”，同时菜单项会被标记为复选，最后再为此菜单项设置快捷键 Alt+V。

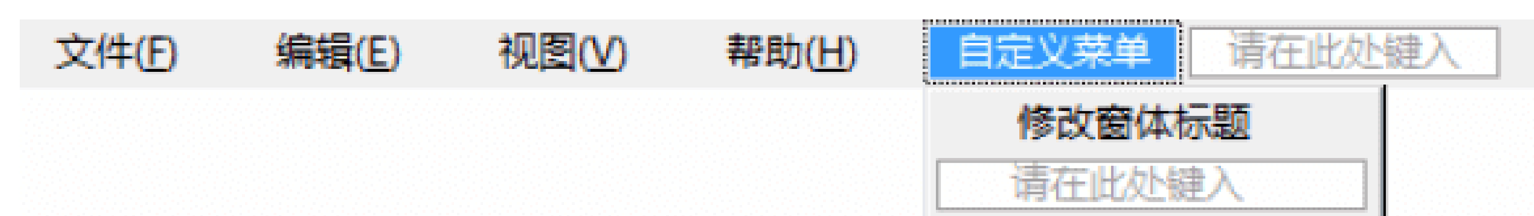


图 6-17　编辑菜单资源

【思路】菜单的命令处理可以参考 6.1.3 小节，菜单的复选处理可以参考 6.1.4 小节，为菜单设置快捷键可以参考 6.1.5 小节。

2．为 MFC 单文档应用程序添加一个工具栏按钮，它的作用同第 1 题添加的子菜单项相同。需要用资源编辑器编辑工具栏资源，再为其添加工具提示“用来修改主窗体的标题”。

【思路】可以参考 6.2.1 小节和 6.2.3 小节所讲解的内容进行操作。

3．在 MFC 单文档应用程序原有的状态栏资源的基础上再“开辟”一块区域，用来显示文本字符串“我添加的状态栏字符串”。

【思路】可以参考 6.3.2 小节所讲解的内容。

第 7 章　使用 Windows 标准控件

为了提高常用代码的复用性，VC 使用控件将常用的诸如用户输入、操作数据等功能封装起来。通常控件放在对话框或工具栏中，分为 3 种：Windows 标准控件、ActiveX 控件和 MFC 支持的其他控件类。本章主要介绍 Windows 标准控件和 ActiveX 控件。

7.1　Windows 标准控件

在 Windows 系统下，系统提供了一组可编程的 Windows 标准控件。VC 将其集成在 IDE 中，读者可以在对话框编辑器中使用拖动的方式直接操作 Windows 标准控件，并且 MFC 对 Windows 标准控件进行了封装。本节将概要地介绍了 Windows 标准控件。

7.1.1　常用 Windows 控件

Windows 控件主要是对界面功能的封装，使用它可以将常用的输入功能非常简单地添加到对话框应用程序中，而不需要开发人员把精力集中在界面开发上。如表 7-1 所示，列出了常用的 Windows 控件。

表 7-1　常用的Windows控件

控　　件	MFC 类	说　　明
按钮控件	CButton	按钮控件，可以产生单击事件；也可以扩展为复选框、单选框按钮分组框
组合框	CComboBox	编辑框和列表框的组合
日期时间选择框	CDateTimeCtrl	可以选择指定的日期或时间值
编辑控件	CEdit	文本输入框
扩展组合框	CComboBoxEx	可以显示图片的组合框
标题控件	CHeaderCtrl	列头按钮，用于控制文本的显示
热键控件	CHotKeyCtrl	用户热键控件
图像列表	CImageList	图像列表，用于管理一组图标或位图
列表视图控件	CListCtrl	显示带有图标的文本列表
列表控件	CListBox	显示字符串列表的控件
月历	CMonthCalCtrl	显示日期信息的控件
进度条控件	CProgressCtrl	指示操作过程进度的控件
控件工具栏	CRebarCtrl	包含子控件的工具栏
扩展编辑框	CRichEditCtrl	带有段落和字符格式的编辑框
滚动条	CScrollBar	对话框中用于滚动查看的滚动条
滑块控件	CSliderCtrl	用于定位选项位置的滑块控件

续表

控　　件	MFC 类	说　　明
微调按钮	CSpinButtonCtrl	用于定量增加或定量减少的微调按钮
静态控件	CStatic	标记其他控件的文本控件
状态栏控件	CStatusBarCtrl	显示状态信息的状态栏
工具栏	CToolBarCtrl	包含命令按钮的工具栏
工具提示	CToolTipCtrl	小的弹出对话框，用于描述工具栏按钮或其他工具功能的控件
树形视图控件	CTreeCtrl	显示树形列表项的属性视图控件

上表中列出了常用的 Windows 控件，从 7.2 节开始，会具体介绍其中常用的 Windows 标准控件。

7.1.2　使用对话框编辑器创建控件

在 Visual Studio 2010 中，有两种方式创建控件。一种是使用 new 关键字在堆上创建控件对象。使用此种方式创建控件，需要在退出程序时，调用 delete 关键字销毁对象。

另一种方式是在对话框编辑器中创建控件对象。此种方式支持所见即所得，并且在程序退出时，系统会自动销毁 Windows 控件。因此，在 Visual Studio 2010 中建议使用对话框编辑器创建控件，具体步骤如下。

（1）打开对话框资源编辑器，如图 7-1 所示。

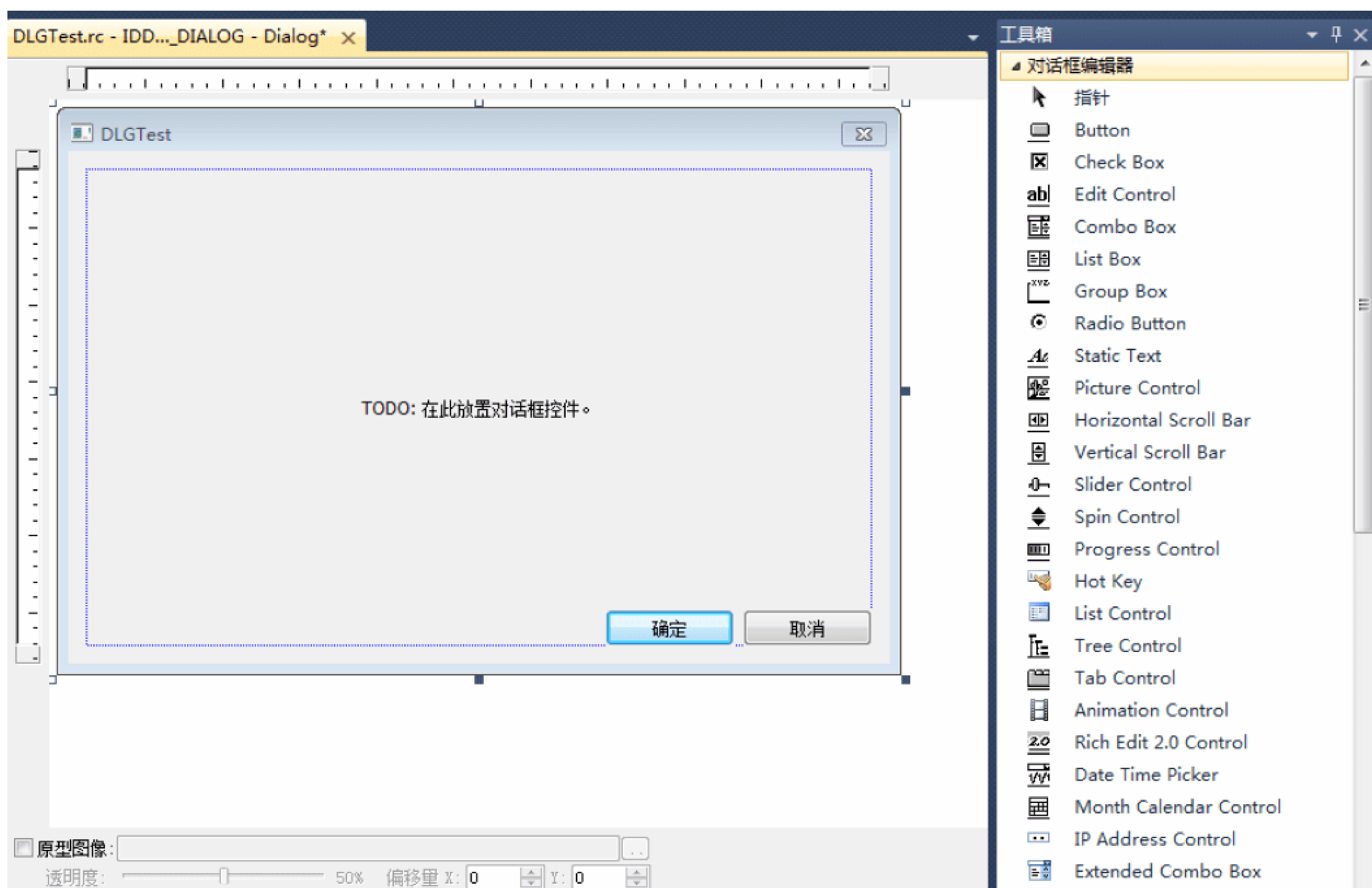


图 7-1　对话框资源编辑器

（2）在图 7-1 中右边的“工具箱”中，单击需要添加的控件，将鼠标移动到对话框主窗体中。单击鼠标添加控件，或是按下需要添加的控件，移动鼠标到对话框主窗体的合适位置，松开鼠标添加控件。

（3）单击要设置大小的控件，将鼠标移动到控制点进行拖动，直到控件大小符合需要。

（4）右击已经添加的控件，单击“属性”命令，弹出属性对话框，在 ID 组合框中指定控件 ID，在 Caption 文本框中指定控件显示的文本，如图 7-2 所示。

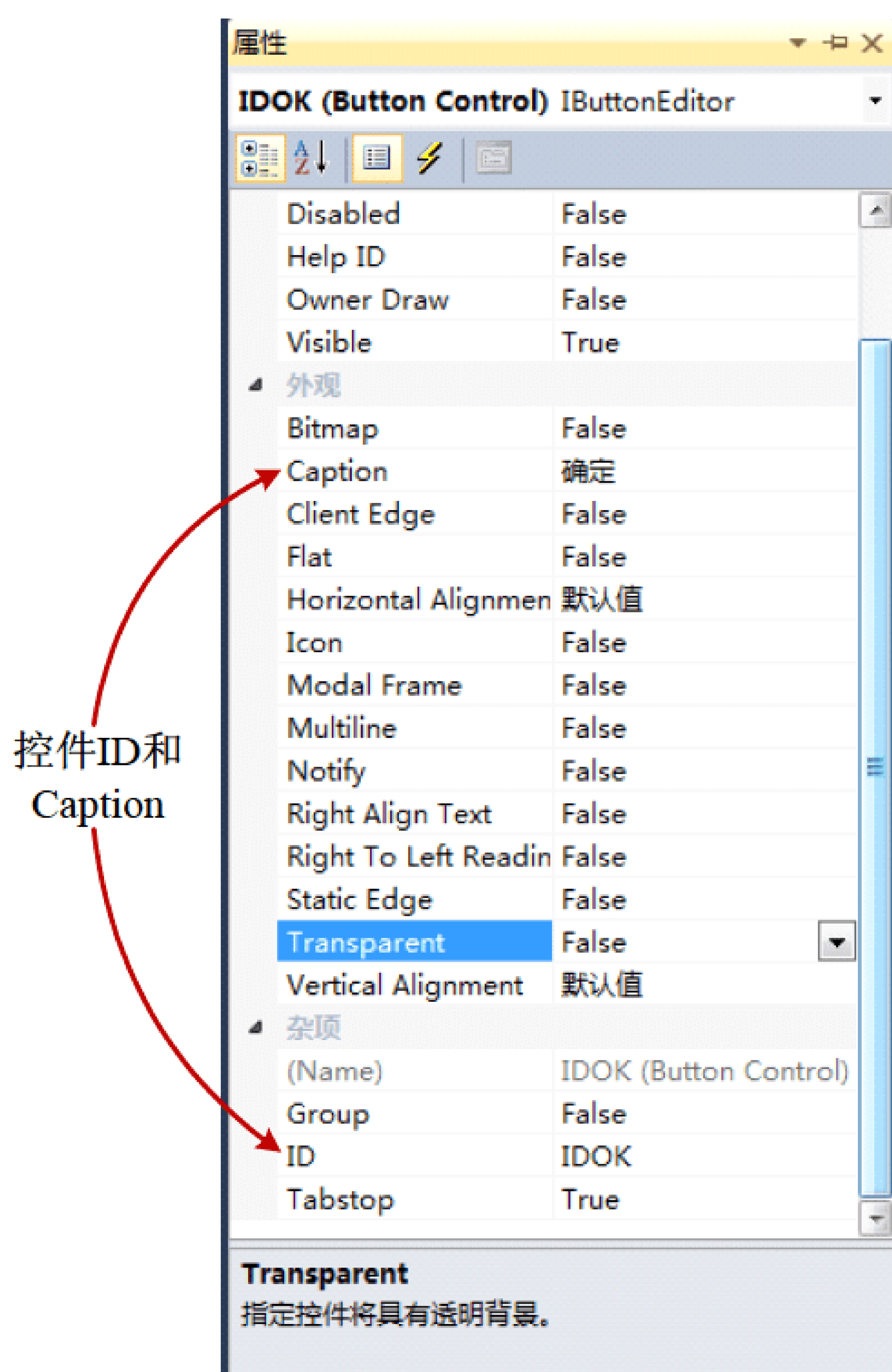


图 7-2　指定控件 ID 和 Caption

（5）重复第（2）～（4）步，依次添加所有需要的控件。这样就可以使用对话框编辑器向程序中添加控件了。

7.1.3　控件类的基类 CWnd

CWnd 类是 MFC 中所有窗体类包括控件类的基类，与 Windows 对话框是不同的，但是又有许多相似之处。CWnd 对象由构造函数和析构函数创建和销毁。另一方面，Windows 对话框是 Windows 内部的数据结构，由 Create()成员函数创建并由 CWnd 的虚析构函数销毁。

CWnd 类和消息映射机制隐藏在 WndProc()函数中。当 Windows 通知消息到来时，会自动路由到相应的 CWnd 的 OnMessage()函数中，可以通过重写这些 OnMessage()成员函数在派生类中处理特殊的成员消息。CWnd 类是可以继承的，使用 CWnd 可以派生子控件，然后向派生类中增加成员变量存储程序数据，在派生类中实现消息处理成员函数和消息映射。创建派生自 CWnd 的子控件分为以下两步。

（1）调用 CWnd 的构造函数构造 CWnd 对象。

（2）调用 Create()成员函数创建子对话框，并将其附加到 CWnd 对象中。

当终止子控件时，销毁 CWnd 对象，或调用 DestroyWindow()成员函数移除控件，并

销毁数据结构。

在 MFC 中，有许多类继承自 CWnd 类，包括表示框架的 CFrameWnd 类，表示多文档框架的 CMDIFrameWnd 类，表示多文档子框架的 CMDIChildWnd 类、表示视图的 CView 类和表示对话框的 CDialog 类，还包括表示控件的类，如 CButton 和 CEdit 等。后面会陆续介绍这些类。

7.1.4　控件的消息及其处理

当发生事件时如用户在控件中输入数据，标准控件会发送通知消息给父对话框，此时，控件就作为对话框的子窗口处理。应用程序就是根据这些通知消息确定用户想要执行的操作的。要处理控件发送给父对话框的通知消息，需要在父类中为每条消息增加一条消息映射条目和消息处理成员函数。一般控件都放置在对话框（派生自 CDialog 类的对话框对象）中。要处理控件消息，则在父对话框类中为每条消息条目添加以下代码。

```
ON_Notification( id, memberFunction )       //消息映射
```

此处 id 表示发送消息的控件 ID，memberFunction 是父类中处理此消息的成员函数名称。而父类中的控件消息处理函数的形式如下：

```
afx_msg void memberFunction ( );            //消息处理函数声明
```

以下代码在源文件中增加对 ID 为 IDC_BUTTON_TEST 的按钮的单击事件（BN_CLICKED）的消息映射条目，表示此按钮的单击事件发生时，执行 OnBnClickedButtonTest()函数。

```
01  BEGIN_MESSAGE_MAP(CButtonSampleDlg, CDialogEx)
02      ON_WM_SYSCOMMAND()
03      ON_WM_PAINT()
04      ON_WM_QUERYDRAGICON()
05      ON_BN_CLICKED(IDC_BUTTON_TEST,
06                    &CButtonSampleDlg::OnBnClickedButtonTest)
07  END_MESSAGE_MAP()
```

在类的头文件中增加对消息处理函数的声明，代码如下：

```
01      afx_msg void OnBnClickedButtonTest();
```

上面代码声明了 OnBnClickedButtonTest()函数。下面代码在类的源文件中定义了此函数，执行的功能就是弹出对话框提示用户单击了此按钮。

```
01  void CButtonSampleDlg::OnBnClickedButtonTest()
02  {
03      //TODO: 在此添加控件通知处理程序代码
04      MessageBox("单击了按钮", "提示");
05  }
```

从上面的代码中可以看出，对控件消息的处理在代码中主要有 3 步，分别是添加消息映射、声明消息处理函数和定义消息处理函数。程序的运行效果如图 7-3 所示。

为了方便操作，Visual Studio 2010 提供了消息处理向导，使用此向导，上面这 3 步的工作可以以可视的方式完成。具体步骤如下：



图 7-3　控件的消息处理示例

（1）在 Visual Studio 2010 中，通过 Ctrl+Shift+X 快捷键或单击“项目”|“类向导”命令，打开“MFC 类向导”对话框。选择“命令”选项卡，如图 7-4 所示。

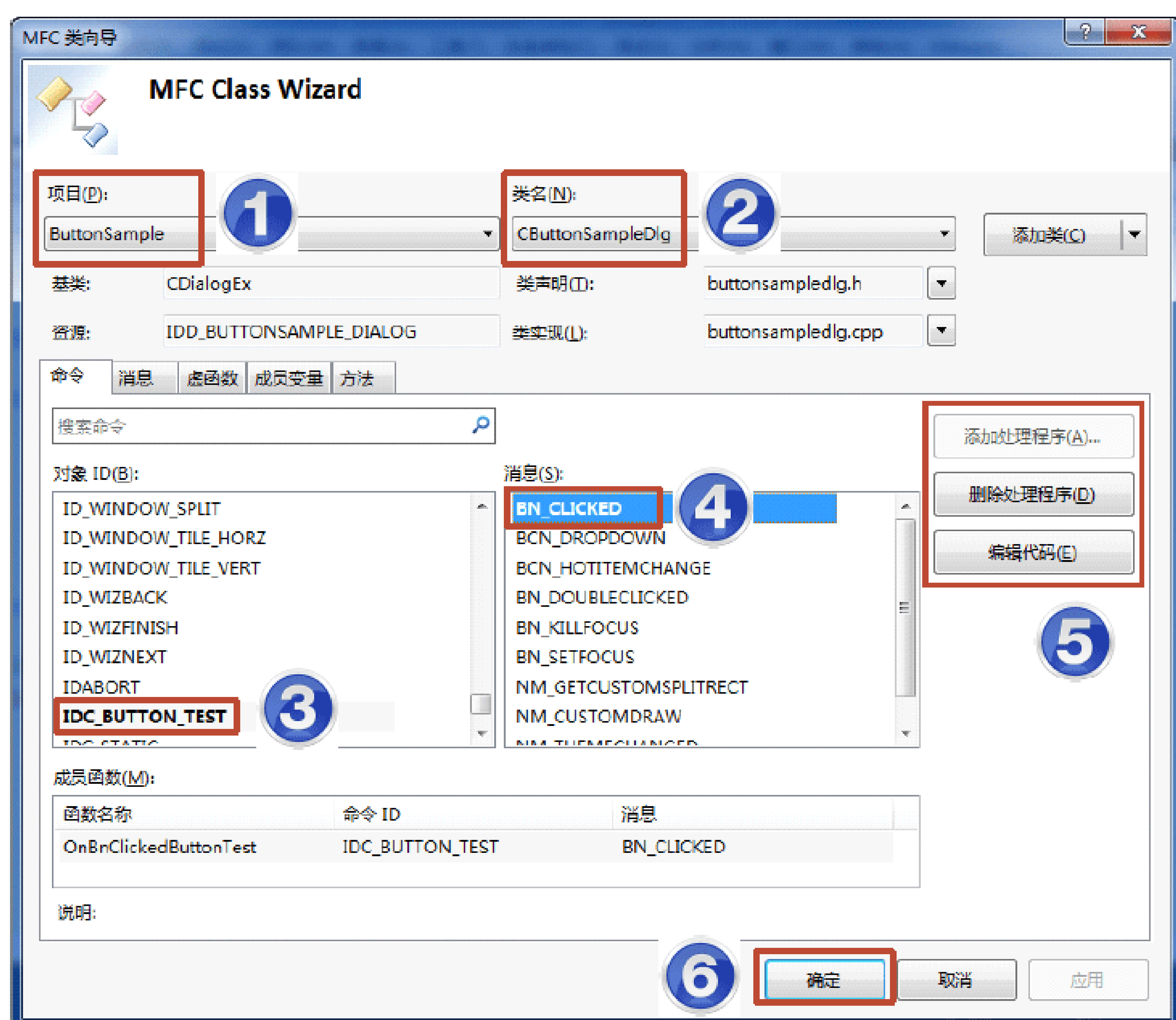


图 7-4　“MFC 类向导”对话框

（2）在“项目”下拉列表框中选择当前操作的工程。在“类名”下拉列表框中选择要处理控件消息所在的父类。在“对象 ID”列表框中选择要处理消息的发送控件。在“消息”列表框中选择要处理的消息。单击“添加处理程序”按钮增加消息处理函数，或单击“删除处理程序”按钮删除消息处理函数。而“成员函数”列表框中列出了当前类中所有的消息映射条目。

（3）单击“编辑代码”按钮跳转到代码编辑器中，编辑消息处理函数中的代码。

（4）单击“确定”按钮退出对话框。

本章后面小节中使用控件的方法与本节介绍的方法相同，就不再重复，而只介绍各种不同控件的不同用法。

7.1.5　创建控件对象

在程序代码中要操作控件，需要使用控件对应的控件对象进行操作。创建控件对象的

步骤如下：

（1）按照 7.1.4 小节介绍的方法打开“MFC 类向导”对话框。选择“成员变量”选项卡，如图 7-5 所示。

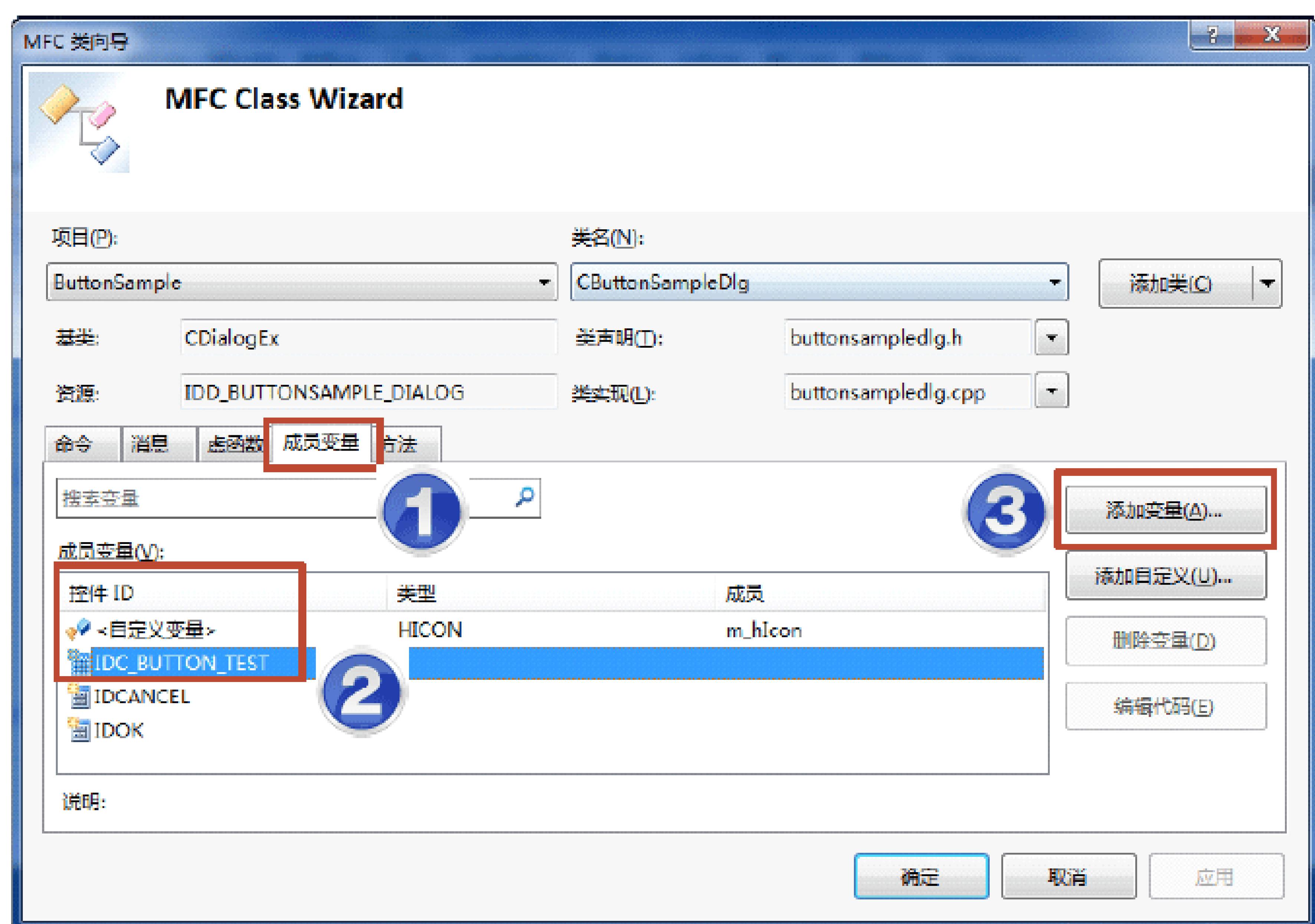


图 7-5　成员变量向导

（2）在图 7-5 中，“控件 ID”列表框列出了当前类中所有的控件的 ID 以及与其对应的控件成员和类型。选择要增加变量的控件，单击“添加变量”按钮，打开如图 7-6 所示的对话框。

（3）在图 7-6 中的“成员变量名称”文本框中输入变量名，在“类别”下拉列表框中选择要增加的变量种类。如果要增加控件值变量，则在“类别”下拉列表框中选择 Value 选项，如果要增加控件对象变量，则在“类别”下拉列表框中选择 Control 选项。在“变量类型”下拉列表框中选择要增加的变量的类型。单击“确定”按钮，即完成了控件成员变量的增加。



图 7-6　增加成员变量

（4）在成员变量向导中，单击“删除变量”按钮，可以删除控件对应的变量。

（5）重复上述第（2）～（4）步，直到定义完所有的控件变量，单击“确定”按钮。

7.2　按　　钮

在 Windows 程序中最常见的操作就是单击某个对象触发一组操作，实现此功能的控件称为按钮控件。按钮控件分为很多种，包括实现单击事件的按钮控件、实现多项选择的复选框控件和实现单选的单选按钮控件。本节将介绍按钮控件的使用。

7.2.1　按钮简介

按钮控件其实是一个小的窗口，有两个状态——按下和抬起。按钮控件可以单独使用，

也可以分组使用。既可以显示文本使用，也可以不显示任何文本。按钮控件的一个典型应用就是当用户单击按钮控件时，改变其外观显示。按钮控件分为复选框、单选按钮和命令按钮。

7.2.2　按钮类 CButton

在 VC 中，使用 CButton 类对象表示按钮控件，它继承自对话框类 CWnd。VC 还提供了一个继承自 CButton 类的 CBitmapButton 类支持图像按钮控件，其提供了一组独立的位图分别表示按钮按下、按钮抬起、按钮选中和按钮不可用这 4 种不同的状态。

7.2.3　按钮的属性与消息

按钮控件没有特殊的属性。常用的消息主要有以下两种。

- ON_BN_CLICKED 消息：当用户单击按钮控件时，发送给父窗体此消息。
- ON_BN_DOUBLECLICKED 消息：当用户双击按钮控件时，发送给父窗体此消息。

一般情况下，处理 ON_BN_CLICKED 消息，执行用户单击控件时的处理。

7.2.4　设定和获取按钮状态

对于单选按钮和复选框，有两个按钮状态，分别是选择和未选择。对于单选按钮，用黑色圆圈表示选择；对于复选框，在方框中有个对勾表示选择。按钮控件的状态可以通过 CButton 类的 4 个函数使用，如下所述。

- GetState()函数：获取单选按钮控件或复选框控件的当前状态。此函数主要是针对单选按钮和复选框而言。如果返回值是 0，表示按钮没有选择；如果是 1，则表示按钮被选择；如果是 2，表示是中间状态；如果是 4，表示当前按钮被用户按下，并高亮显示；如果是 8，表示按钮获得输入焦点。
- SetState()函数：设置按钮控件是否高亮显示。
- GetCheck()函数：返回单选按钮控件或复选框控件的当前状态。返回值为 0，表示没有选择按钮控件；返回值为 1，表示选择了按钮控件；返回值为 2，表示按钮控件处于中间状态。
- SetCheck()函数：设置单选按钮控件或复选按钮控件的当前状态。传入值为 0，表示没有选择按钮控件；传入值为 1，表示选择按钮控件；传入值为 2，表示设置按钮控件处于中间状态。

7.3　静态控件与编辑控件

静态控件是常用的信息提示控件，编辑控件是常用的用户输入控件。这两个控件在 Windows 应用程序中是应用非常频繁的控件，创建与使用的方法与 Windows 标准控件的使用方法是一样的。本节将介绍静态控件和编辑控件所特有的编程属性和功能。最后，以一个实例演示如何使用静态控件和编辑控件。

7.3.1　创建与使用静态控件

创建静态控件的方法与 7.1 节中介绍的方法类似。只是在创建时，在控件工具栏中选择其中的静态控件。一般是在设计时，输入将要显示的内容，但是此时不处理消息。当使用静态控件时，在添加的静态控件上右击，在弹出的快捷菜单中选择“属性”命令，弹出静态控件的“属性”对话框，如图 7-7 所示。

7.3.2　静态控件类 CStatic

CStatic 类提供 Windows 静态控件的功能。静态控件显示文本字符串、矩形、图标、光标、位图或增强型图元文件。可以用作标记、分组或分隔其他控件。通常情况下，静态控件不接收输入也不提供输出。但是如果使用 SS_NOTIFY 样式创建，则可以通知父窗体鼠标单击的事件。如图 7-8 所示，设置 Notify 选项为 True，即可处理静态控件的鼠标单击事件。

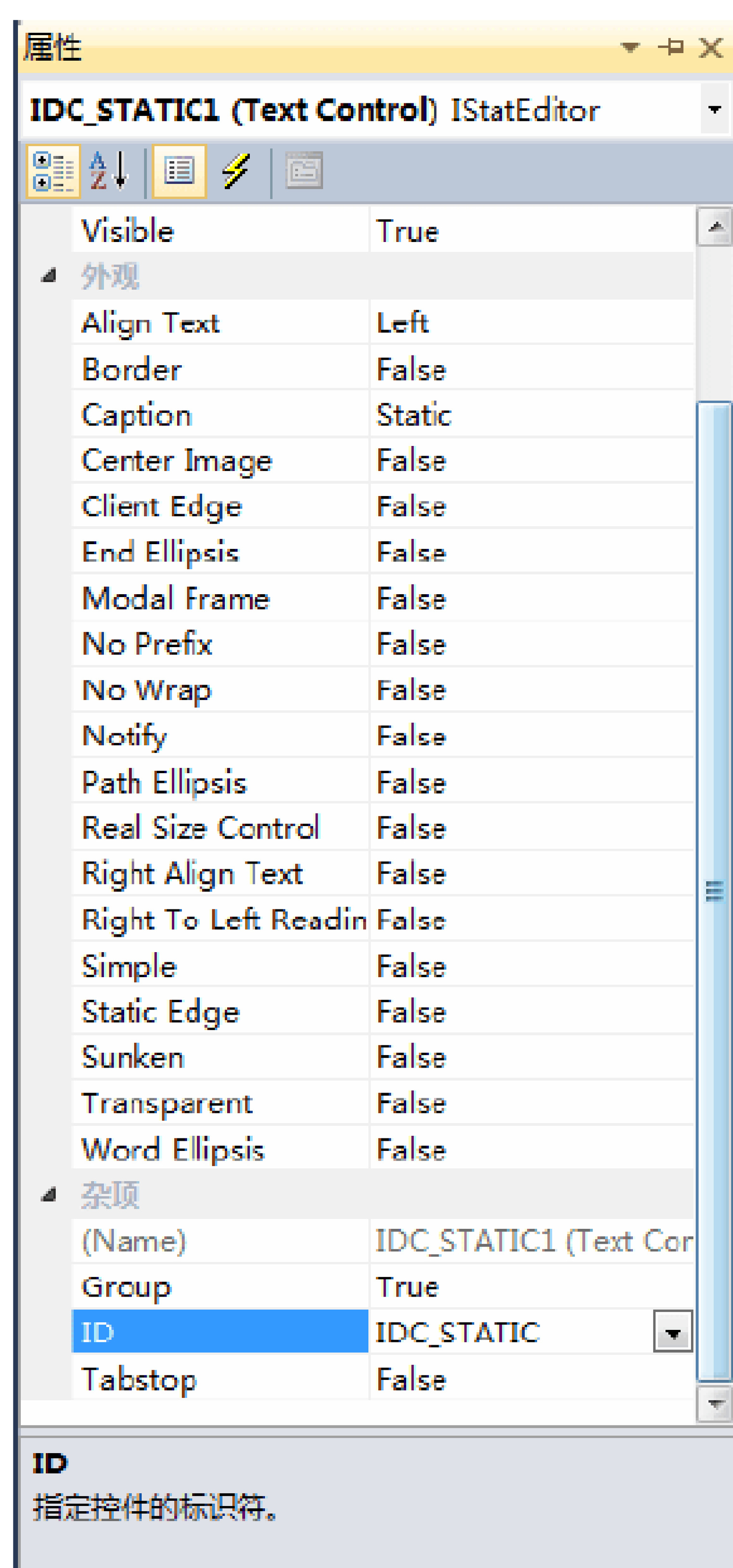


图 7-7　静态控件的使用

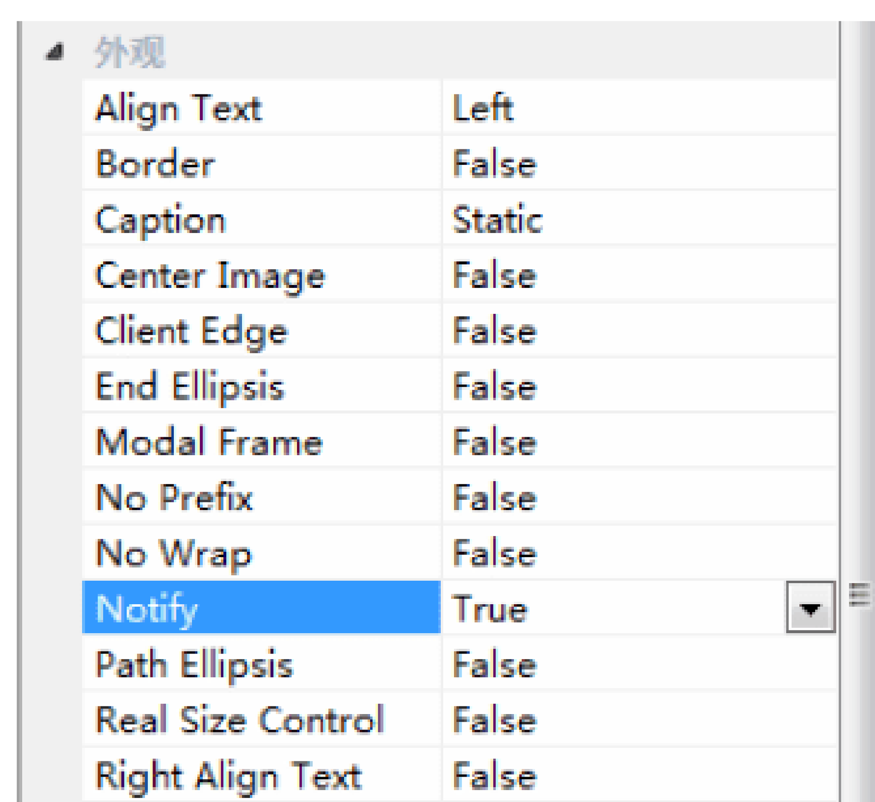


图 7-8　静态控件的消息支持选项

CStatic 类除了提供基本的静态控件的操作外，还提供下面的函数可以操作静态控件的设置，如表 7-2 所示。

表 7-2　CStatic类的成员函数

成 员 函 数	功　　能
SetBitmap()	指定在静态控件上显示的位图
GetBitmap()	获取在静态控件上显示的位图的句柄
SetIcon()	指定在静态控件上显示的图标
GetIcon()	获取在静态控件上显示的图标的句柄

续表

成 员 函 数	功　　能
SetCursor()	指定在静态控件上显示的光标
GetCursor()	获取在静态控件上显示的光标的句柄
SetEnhMetaFile()	指定在静态控件上显示的增强型元文件
GetEnhMetaFile()	获取在静态控件上显示的增强型元文件的句柄

7.3.3　创建编辑控件

编辑控件是一个用于输入文本的长方形子对话框，可以提供用户与程序之间的数据交互。它的创建过程与 Windows 控件的创建过程类似。

7.3.4　编辑控件类 CEdit

CEdit 类提供对话框编辑控件功能。从 CWnd 类中继承有用的功能函数。使用 CWnd 类的 SetWindowText()函数和 GetWindowText()函数，可以从 CEdit 对象中设置和接收文本。如果编辑控件支持多行功能，则可以调用 CEdit 的 GetLine()、SetSel()、GetSel()和 ReplaceSel()成员函数获取或设置控件的部分文本。表 7-3 列出了 CEdit 类的常用成员函数。

表 7-3　CEdit类的常用成员函数

成 员 函 数	功　　能
CanUndo()	确定编辑控件是否可以撤销
GetLineCount()	获取多行编辑器中当前的行数
GetModify()	确定编辑控件中的内容是否被修改过
SetModify()	设置或清除编辑控件修改标记
GetRect()	获取编辑控件的矩形框
GetSel()	获取编辑控件当前选择的内容
GetHandle()	获取多行控件分配的内存句柄
SetHandle()	设置多行控件使用的本地内存的句柄
SetMargins()	设置 CEdit 类的左边和右边边白
GetMargins()	获取 CEdit 类的左边和右边边白
SetLimitText()	设置 CEdit 类中可以存放的最大文本数
GetLimitText()	获取 CEdit 类中可以存放的最大文本数
GetLine()	获取编辑控件指定行的内容
GetPasswordChar()	获取当编辑控件作为密码控件时，显示的字符
ReplaceSel()	使用指定文本替换编辑控件当前选中的内容
SetPasswordChar()	设置当编辑控件作为密码控件时，显示的字符
SetSel()	选在编辑控件中的指定范围
SetReadOnly()	设置编辑控件为只读控件
Undo()	撤销最后一次操作
Clear()	删除编辑控件中当前选择的内容
Copy()	将当前选择的编辑控件中的内容复制到剪贴板
Cut()	将当前选择的编辑控件中的内容剪切到剪贴板
Paste()	粘贴当前剪贴板中的内容到编辑控件中

7.3.5　编辑控件的消息

常用的编辑控件消息有以下几个。

- ON_EN_CHANGE 消息：当用户修改编辑控件中的内容时，发送此消息。与 EN_UPDATE 通知消息不同，当对话框更新显示时，才会发送此通知消息。
- ON_EN_ERRSPACE 消息：当编辑控件不能分配足够的内存处理特殊请求时发送此消息。
- ON_EN_HSCROLL 消息：当用户单击编辑控件的水平滚动条时，在屏幕更新前，编辑控件向父对话框发送此消息。
- ON_EN_KILLFOCUS 消息：当编辑控件失去输入焦点时发送此消息。
- ON_EN_MAXTEXT 消息：当前输入框中的内容超过编辑控件的指定最大字符数时，触发此消息，并且会将多余的内容截除。当编辑控件没有水平自动滚动条，而当前在编辑框中输入的内容超过编辑控件的宽度时，也会发送此消息。当编辑控件没有垂直自动滚动条，而当前在编辑框中输入的内容超过编辑控件的高度时，也会发送此消息。
- ON_EN_SETFOCUS 消息：当编辑控件接收到输入焦点时，发送此消息。
- ON_EN_UPDATE 消息：编辑控件格式化完文本，但是还没有在屏幕上显示前，发送此消息。
- ON_EN_VSCROLL 消息：当用户单击编辑控件的垂直滚动条时，在屏幕更新前，编辑控件向父对话框发送此消息。

7.3.6　编辑控件的应用实例

本小节演示静态控件和编辑控件的各项功能的实现。类的头文件代码如下：

```
01  class CStaticAndEditSampleDlg : public CDialog     //对话框类声明
02  {
03  public:
04      void WriteLog(CString message, CString title); //记录日志函数声明
05      CStaticAndEditSampleDlg(CWnd* pParent = NULL); //标准构造函数
06  //{{AFX_DATA(CStaticAndEditSampleDlg)
07      enum { IDD = IDD_STATICANDEDITSAMPLE_DIALOG };
08      CEdit    m_editTestScroll;         //带滚动条的编辑控件对应的对象
09      CEdit    m_editTest;               //测试编辑控件对应的对象
10      CStatic m_staticLog;               //日志静态框
11  //}}AFX_DATA
12  //{{AFX_VIRTUAL(CStaticAndEditSampleDlg)
13  protected:
14      virtual void DoDataExchange(CDataExchange* pDX);//DDX/DDV 支持
15  //}}AFX_VIRTUAL
16  protected:
17      HICON m_hIcon;                     //图标变量
18  //{{AFX_MSG(CStaticAndEditSampleDlg)             //消息映射
19      virtual BOOL OnInitDialog();     //初始化对话框函数声明
20      afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
```

```
21                                              //系统命令函数声明
22      afx_msg void OnPaint();                 //重绘函数声明
23      afx_msg HCURSOR OnQueryDragIcon();      //查询拖放图标函数声明
24      afx_msg void OnChangeEditTest();        //改变编辑框内容函数声明
25      afx_msg void OnErrspaceEditTest();      //擦除编辑框内容函数声明
26      afx_msg void OnKillfocusEditTest();     //编辑框失去焦点函数声明
27      afx_msg void OnMaxtextEditTest();       //编辑框内容达到最大值函数声明
28      afx_msg void OnSetfocusEditTest();      //设置编辑框焦点函数声明
29      afx_msg void OnUpdateEditTest();        //更新编辑框函数声明
30      afx_msg void OnHscrollEditTestScroll(); //水平滚动条滚动事件函数声明
31      afx_msg void OnVscrollEditTestScroll(); //垂直滚动条滚动事件函数声明
32      afx_msg void OnButtonGetedittext();     //设置文本内容事件函数声明
33      afx_msg void OnButtonSetedittext();     //获取文本内容事件函数声明
34      afx_msg void OnButtonGetline();     //获取文本行内容事件函数声明
35      afx_msg void OnButtonGetsel();      //获取选择的文本内容事件函数声明
36      afx_msg void OnButtonSetsel();      //设置选择的文本内容事件函数声明
37      afx_msg void OnButtonReplacesel();  //替换选择的文本内容事件函数声明
38      afx_msg void OnStaticTest();        //静态控件测试事件函数声明
39  //}}AFX_MSG
40      DECLARE_MESSAGE_MAP()               //结束消息映射
41  };
```

上面的代码是 CStaticAndEditSampleDlg 类的声明头文件。其中定义了两个 CEdit 控件和一个 CStatic 控件变量。m_editTest 控件变量表示左边的编辑控件，用于测试编辑控件的各种消息。m_editTestScroll 控件变量表示右边的编辑控件，用于测试编辑控件的单击水平滚动条和垂直滚动条的消息。WriteLog() 函数用于向日志静态框中写日志。在//{{AFX_MSG(CStaticAndEditSampleDlg)和//}}AFX_MSG 之间，声明了各个消息处理函数。其中大部分代码是由应用向导和类向导生成的。实现类的源文件代码如下：

```
01  //对话框初始化函数
02  CStaticAndEditSampleDlg::
03      CStaticAndEditSampleDlg(CWnd* pParent /*=NULL*/)
04      : CDialog(CStaticAndEditSampleDlg::IDD, pParent)
05  {
06      //{{AFX_DATA_INIT(CStaticAndEditSampleDlg)
07      //}}AFX_DATA_INIT
08      m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);//装载应用程序图标
09  }
10  //数据交换函数
11  void CStaticAndEditSampleDlg::DoDataExchange(CDataExchange* pDX)
12  {
13      CDialog::DoDataExchange(pDX);               //执行基类的数据交换函数
14  //{{AFX_DATA_MAP(CStaticAndEditSampleDlg)       //控件和数据映射对应关系
15      DDX_Control(pDX, IDC_EDIT_TEST_SCROLL, m_editTestScroll);
16      DDX_Control(pDX, IDC_EDIT_TEST, m_editTest);
17      //日志静态框变量声明
18      DDX_Control(pDX, IDC_STATIC_LOG, m_staticLog);
19  //}}AFX_DATA_MAP
20  }
21  BEGIN_MESSAGE_MAP(CStaticAndEditSampleDlg, CDialog)//消息映射表开始
22  //{{AFX_MSG_MAP(CStaticAndEditSampleDlg)
23      ON_WM_SYSCOMMAND()                          //系统命令消息
24      ON_WM_PAINT()                               //重绘消息
25      ON_WM_QUERYDRAGICON()                       //查询拖动图标消息
```

```
26      //编辑框内容改变消息映射
27      ON EN CHANGE(IDC EDIT TEST, OnChangeEditTest)
28      //编辑框内容擦除消息映射
29      ON EN ERRSPACE(IDC EDIT TEST, OnErrspaceEditTest)
30      //编辑框失去焦点消息映射
31      ON EN KILLFOCUS(IDC EDIT TEST, OnKillfocusEditTest)
32      //编辑框内容改变消息映射
33      ON_EN_MAXTEXT(IDC_EDIT_TEST, OnMaxtextEditTest)
34      //编辑框获得焦点消息映射
35      ON EN SETFOCUS(IDC EDIT TEST, OnSetfocusEditTest)
36      //编辑框内容更新消息映射
37      ON_EN_UPDATE(IDC_EDIT_TEST, OnUpdateEditTest)
38      //水平滚动消息
39      ON EN HSCROLL(IDC EDIT TEST SCROLL, OnHscrollEditTestScroll)
40      //垂直滚动消息
41      ON EN VSCROLL(IDC EDIT TEST SCROLL, OnVscrollEditTestScroll)
42      //获取文本消息
43      ON_BN_CLICKED(IDC_BUTTON_GETEDITTEXT, OnButtonGetedittext)
44      //设置文本消息
45      ON BN CLICKED(IDC BUTTON SETEDITTEXT, OnButtonSetedittext)
46      ON_BN_CLICKED(IDC_BUTTON_GETLINE, OnButtonGetline) //获取行
47      //设置选择的内容
48      ON BN CLICKED(IDC BUTTON GETSEL, OnButtonGetsel)
49      //获取选择的内容
50      ON_BN_CLICKED(IDC_BUTTON_SETSEL, OnButtonSetsel)
51      //文本替换
52      ON_BN_CLICKED(IDC_BUTTON_REPLACESEL, OnButtonReplacesel)
53      ON BN CLICKED(IDC STATIC TEST, OnStaticTest)      //信息提示
54  //}}AFX MSG MAP
55  END MESSAGE MAP()
56  //初始化对话框
57  BOOL CStaticAndEditSampleDlg::OnInitDialog()
58  {
59      //调用基类的对话框初始化函数
60      CDialog::OnInitDialog();
61      ASSERT((IDM ABOUTBOX & 0xFFF0) == IDM ABOUTBOX);
62                                                  //判断是否为"关于"命令
63      ASSERT(IDM_ABOUTBOX < 0xF000);
64      CMenu* pSysMenu = GetSystemMenu(FALSE);//获取菜单
65      if (pSysMenu != NULL)                       //判断菜单是否为 NULL
66      {
67          //定义存放菜单名称的字符串变量
68          CString strAboutMenu;
69          strAboutMenu.LoadString(IDS_ABOUTBOX); //装载关于对话框的菜单
70          if (!strAboutMenu.IsEmpty())           //如果关于菜单不为空
71          {
72              pSysMenu->AppendMenu(MF_SEPARATOR);//增加分隔符
73              //增加"关于"菜单命令
74              pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
75                                                  strAboutMenu);
76          }
77      }
78      SetIcon(m_hIcon, TRUE);          //设置大图标
79      SetIcon(m_hIcon, FALSE);         //设置小图标
80      return TRUE;                     //返回 TRUE
81  }
82  void CStaticAndEditSampleDlg::OnSysCommand(UINT nID, LPARAM lParam)
```

```
83                                          //处理系统命令
84  {
85      if ((nID & 0xFFF0) == IDM_ABOUTBOX)
86                                          //判断单击选择的命令是否为“关于”命令
87      {
88          CAboutDlg dlgAbout;             //定义“关于”对话框
89          dlgAbout.DoModal();             //显示“关于”对话框
90      }
91      else        CDialog::OnSysCommand(nID, lParam);
92                                          //如果不是“关于”命令，则处理命令消息
93  }
94  void CStaticAndEditSampleDlg::OnPaint()     //对话框绘制函数
95  {
96      if (IsIconic())                     //判断是否是图标状态
97      {
98          CPaintDC dc(this);              //进行绘制的设备上下文
99          //发送图标绘制背景消息
100         SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);
101         //将图标放置在客户端矩形中间
102         int cxIcon = GetSystemMetrics(SM_CXICON);//获取小图标的 X 长度
103         int cyIcon = GetSystemMetrics(SM_CYICON);//获取小图标的 Y 高度
104         CRect rect;                                 //定义矩形区域
105         GetClientRect(&rect);                       //获取客户区矩形
106         //计算客户区中心点的 X 值
107         int x = (rect.Width() - cxIcon + 1) / 2;
108         //计算客户区中心点的 Y 值
109         int y = (rect.Height() - cyIcon + 1) / 2;
110         dc.DrawIcon(x, y, m_hIcon);                 //绘制图标
111     }
112     else    CDialog::OnPaint();                     //执行基类的绘制函数
113 }
114 //获取拖动图标消息处理函数
115 HCURSOR CStaticAndEditSampleDlg::OnQueryDragIcon()
116 {
117     return (HCURSOR) m_hIcon;                       //返回图标变量
118 }
119 //文本内容改变消息处理函数
120 void CStaticAndEditSampleDlg::OnChangeEditTest()
121 {
122     WriteLog("接收到 ON_EN_CHANGE 消息", "左边的编辑控件");
123 }
124 //文本内容擦除消息处理函数
125 void CStaticAndEditSampleDlg::OnErrspaceEditTest()
126 {
127     WriteLog("接收到 ON_EN_ERRSPACE 消息", "左边的编辑控件");
128 }
129 //失去焦点消息处理函数
130 void CStaticAndEditSampleDlg::OnKillfocusEditTest()
131 {
132     WriteLog("接收到 ON_EN_KILLFOCUS 消息", "左边的编辑控件");
133 }
134 //达到最大文本数消息处理函数
135 void CStaticAndEditSampleDlg::OnMaxtextEditTest()
136 {
137     WriteLog("接收到 ON_EN_MAXTEXT 消息", "左边的编辑控件");
138 }
139 //获取焦点消息处理函数
```

```
140 void CStaticAndEditSampleDlg::OnSetfocusEditTest()
141 {
142     WriteLog("接收到 ON_EN_SETFOCUS 消息", "左边的编辑控件");
143 }
144 void CStaticAndEditSampleDlg::OnUpdateEditTest()//更新消息处理函数
145 {
146     WriteLog("接收到 ON_EN_UPDATE 消息", "左边的编辑控件");
147 }
148 void CStaticAndEditSampleDlg::OnHscrollEditTestScroll()
149                                                     //水平滚动消息处理函数
150 {
151     WriteLog("接收到 ON_EN_HSCROLL 消息", "右边的编辑控件");
152 }
153 void CStaticAndEditSampleDlg::OnVscrollEditTestScroll()
154                                                     //垂直滚动消息处理函数
155 {
156     WriteLog("接收到 ON_EN_VSCROLL 消息", "右边的编辑控件");
157 }
158 //显示日志函数
159 void CStaticAndEditSampleDlg::WriteLog(CString message,
160  CString title)
161 {
162     //获取当前日志静态框的文本内容
163     m_staticLog.SetWindowText(title + "--" + message);
164 }
165 //获取编辑框内容处理函数
166 void CStaticAndEditSampleDlg::OnButtonGetedittext()
167 {
168     CString content;                                //定义编辑框内容字符串
169     m_editTest.GetWindowText(content);              //获取编辑框内容
170     //在弹出对话框中显示获取的编辑框内容
171     MessageBox(content, "获取左边编辑框内容");
172 }
173 //设置编辑框内容处理函数
174 void CStaticAndEditSampleDlg::OnButtonSetedittext()
175 {
176     m_editTest.SetWindowText("您好!这是测试");
177 }
178 //获取编辑框指定行内容的处理函数
179 void CStaticAndEditSampleDlg::OnButtonGetline()
180 {
181     TCHAR content[256];                     //存放内容的字符串变量
182     memset(content, 0x00, sizeof(content));     //初始化字符串数组
183     int iCount = m_editTest.GetLine(1, content, sizeof(content));
184                                                 //获取第 2 行内容
185     if (iCount > 0)
186         MessageBox(content, "GETLINE 获取第 2 行的内容");
187                                                 //显示获取的内容
188     else
189         MessageBox("失败", "GETLINE 获取第 2 行的内容");
190                                                 //显示失败提示
191 }
192 void CStaticAndEditSampleDlg::OnButtonGetsel() //获取选择的内容
193 {
194     int iStart= 0,iEnd = 0;             //定义开始位置和结束位置的变量
195     //获取选择的内容所在的开始位置和结束位置
196     m_editTest.GetSel(iStart, iEnd);
```

```
197     CString log;                        //日志字符串
198     log.Format("选择的内容从第%d 个字符到第%d 个字符", iStart, iEnd);
199                                         //格式化显示选择的位置
200     MessageBox(log, "GetSel");          //显示信息提示
201 }
202 void CStaticAndEditSampleDlg::OnButtonSetsel() //设置选择的内容
203 {
204     m_editTest.SetSel(5, 10, TRUE);                 //设置选择第 6~11 个字符
205     MessageBox("选择编辑控件中的第 6 个字符到第 11 个字符", "SetSel");
206                                                     //显示提示信息
207 }
208 void CStaticAndEditSampleDlg::OnButtonReplacesel() //文本替换命令
209 {
210     m_editTest.ReplaceSel("此处是新替换的内容");
211 }
212 void CStaticAndEditSampleDlg::OnStaticTest()        //静态控件处理函数
213 {
214     MessageBox("如果使用 SS_NOTIFY 创建静态控件，\n 则可以接收单击事件，
215                       \n 此处就是  例子。", "静态控件");
216 }
```

上面代码定义了 CStaticAndEditSampleDlg 类的各个消息处理函数。这里会以消息框或者写入静态框的方式提示接收到消息，具体的消息处理内容需要根据用户的实际需要添加。此处列出完整的文件内容，为了减少篇幅后面会将减少的代码列出，向导生成的代码会省略掉，此实例的主窗体如图 7-9 所示。

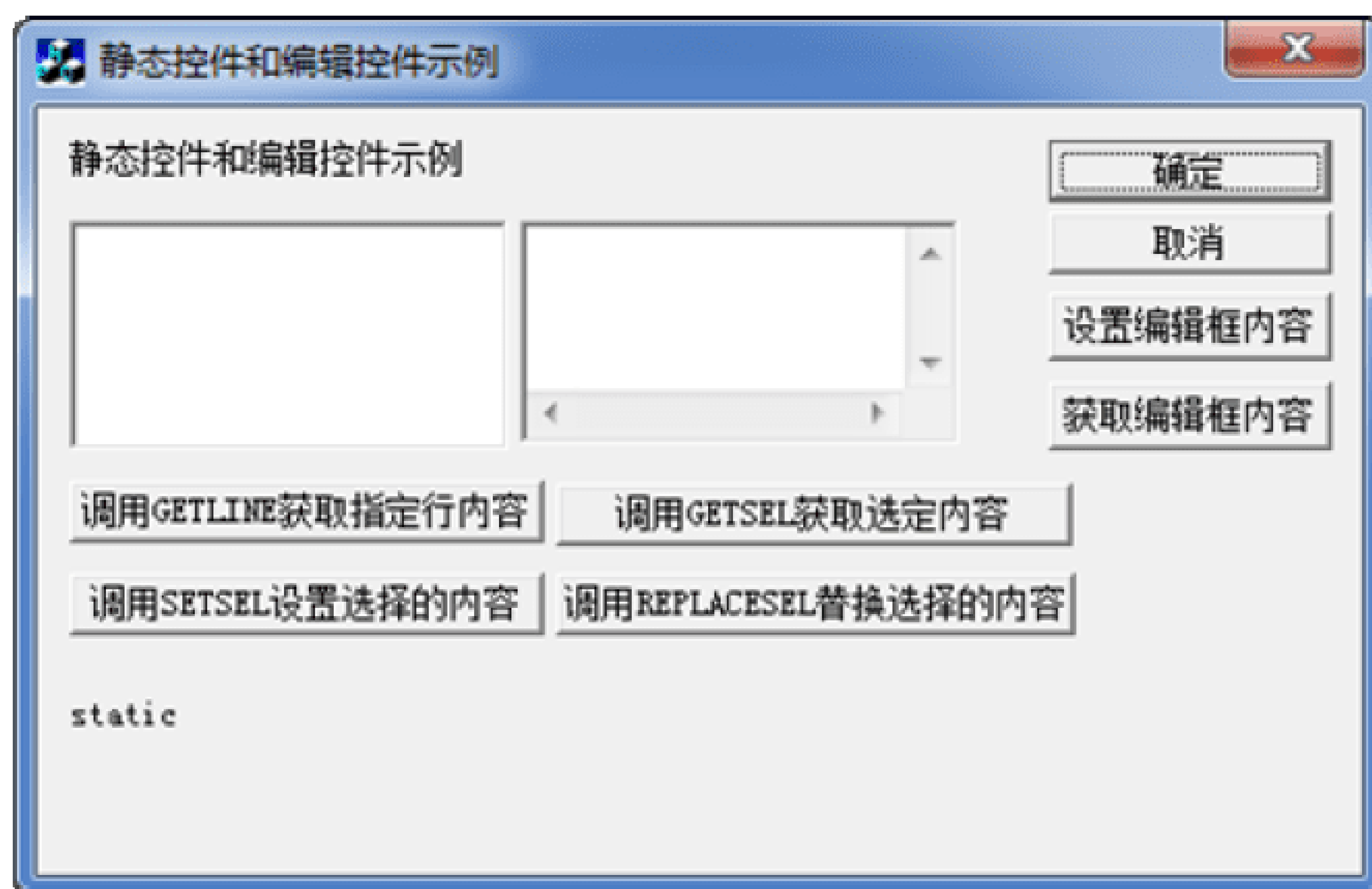


图 7-9　静态控件和编辑控件实例的主窗体

7.4　单选按钮和复选框

单选按钮和复选框都是特殊的按钮控件。单选按钮控件允许用户在一组选项中，选择其中的一项；复选框允许用户在一组选项中，选择其中的多项。单选按钮控件和复选框控件都有分组的概念，尤其是单选按钮控件，是指在同组中只能选择一项。本节将介绍单选按钮控件和复选框控件的使用。

7.4.1　单选按钮控件的创建

创建单选按钮控件的方法与 7.1.2 小节中介绍的方法类似。只是在创建时，在控件工具栏中选择其中的单选按钮控件。要注意的是，因为单选按钮是多选一的控件，所以需要添加多个单选按钮。而一个界面上，会遇到一组以上的单选按钮控件。这时，需要将其分组。操作过程如下：

（1）将各个单选按钮控件的 Tab 键顺序按照分组设置，即同一组的单选按钮控件的 Tab 键顺序需要在一起。方法是选择 Ctrl+D 快捷键或选择“格式”|“Tab 键顺序”命令，弹出如图 7-10 所示的界面。在此界面中，按照顺序依次单击各个控件。设置完 Tab 键顺序后，单击其他区域完成设置。

（2）在图 7-11 中，选择每个分组中的第一个单选按钮的 Group 属性、Tab stop 属性以及 Auto 属性。如此例中“红色”代表的单选按钮和“男”代表的单选按钮都需要设置这 3 个属性为 True。



图 7-10　对话框控件的 Tab 键的设置

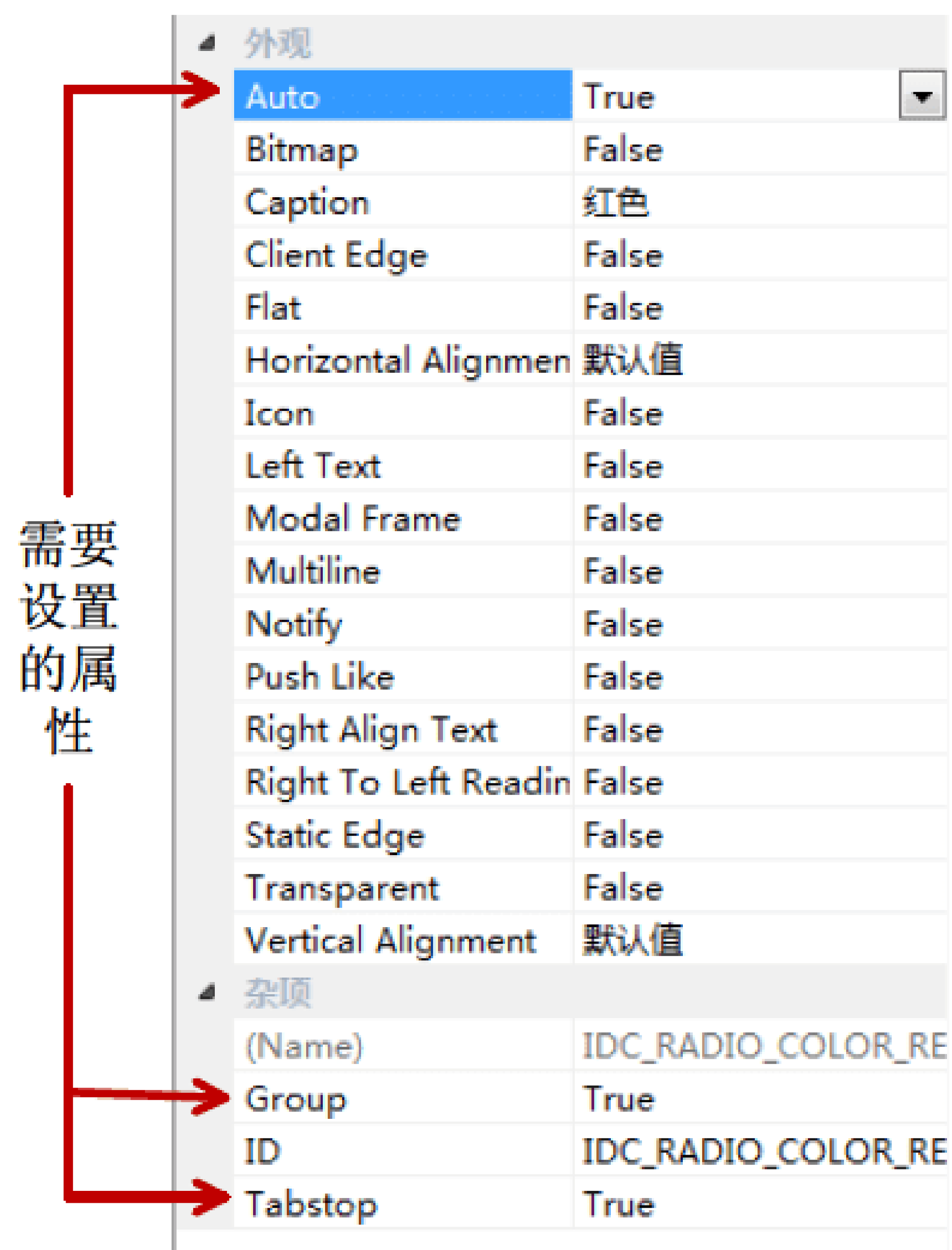


图 7-11　单选按钮的属性设置

（3）其余的单选按钮设置 Tab stop 属性以及 Auto 属性即可。这样，就将上面的 5 个单选按钮分为两组，一组是颜色单选组，一组是性别单选组。

7.4.2　单选按钮控件的消息

单选按钮常用的消息主要有以下两个。

- ON_BN_CLICKED 消息：当用户单击单选按钮控件时，发送给其父窗体此消息。
- ON_BN_DOUBLECLICKED 消息：当用户双击单选按钮控件时，发送给其父窗体此消息。

一般情况下，处理 ON_BN_CLICKED 消息，执行用户选择某个单选按钮时需要执行的操作。

7.4.3　复选框控件的创建

创建复选框控件的方法与 7.1.2 小节中介绍的方法类似。只是在创建时，在控件工具栏中选择其中的复选框控件。

7.4.4　复选框控件的消息

复选框常用的消息主要有以下两个。

- ON_BN_CLICKED 消息：当用户单击复选按钮控件时，发送给其父窗体此消息。
- ON_BN_DOUBLECLICKED 消息：当用户双击复选按钮控件时，发送给其父窗体此消息。

一般情况下，处理 ON_BN_CLICKED 消息，执行用户选择某个复选框时需要执行的操作。

7.4.5　单选按钮控件和复选框控件的实例

在 RadioAndCheckBoxSample 示例中，演示了如何使用单选按钮控件和复选框控件。获取这两种控件的选择状态的代码如下：

```
01  void CRadioAndCheckBoxSampleDlg::OnButtonGetstate()
02  {
03      //TODO: Add your control notification handler code here
04      //颜色选择
05      UINT iColor[] = {IDC_RADIO_COLOR_RED, IDC_RADIO_COLOR_GREEN,
06                                        IDC_RADIO_COLOR_BLUE};
07      CString sColor[] = {"红色", "绿色", "蓝色"};
08      CString sResultColor;
09      CButton *pBtn=NULL;
10      for(int i=0; i<3; i++)
11      {
12          pBtn = (CButton*)GetDlgItem(iColor[i]);
13          if(!pBtn)
14              continue;
15
16          if(pBtn->GetCheck() == 1)
17              sResultColor = "颜色选择: " + sColor[i];
18      }
19
20      //性别选择
21      UINT iSex[] = {IDC_RADIO_SEX_MALE, IDC_RADIO_SEX_FEMALE};
22      CString sSex[] = {"男", "女"};
23      CString sResultSex;
24      for(int i=0; i<2; i++)
25      {
26          pBtn = (CButton*)GetDlgItem(iSex[i]);
27          if(!pBtn)
28              continue;
29
30          if(pBtn->GetCheck() == 1)
```

```
31              sResultSex = "\n 性别选择: " + sSex[i];
32       }
33
34       //喜好选择
35       UINT iLike[] = {IDC_CHECK_LIKE_BOOK, IDC_CHECK_LIKE_MUSIC,
36                       IDC_CHECK_LIKE_SPORT,IDC_CHECK_LIKE_DANCE};
37       CString sLike[] = {"读书", "听音乐", "运动", "跳舞"};
38       CString sResultLike = "\n 喜欢: ";
39       for(int i=0; i<4; i++)
40       {
41          pBtn = (CButton*)GetDlgItem(iLike[i]);
42          if(!pBtn)
43              continue;
44
45          if(pBtn->GetCheck() == 1)
46              sResultLike =  sResultLike + sLike[i] + "、";
47       }
48
49       //总结输出
50       MessageBox(sResultColor +  sResultSex + sResultLike, "选择结果");
51   }
```

上面的代码依次获取了用户选择的颜色、性别和爱好。其中颜色和性别是单选按钮，爱好是复选框。在获取选项选择时，首先将选项按钮的 ID 号和名称分别存放在数组中，然后通过 for 循环依次判断每个选项是否选择了，如果选择了，就将选项对应的名称存入对应的字符串变量，最后将这 3 项的选择结果值组合起来，以消息框的形式提示给用户，运行效果如图 7-12 所示。

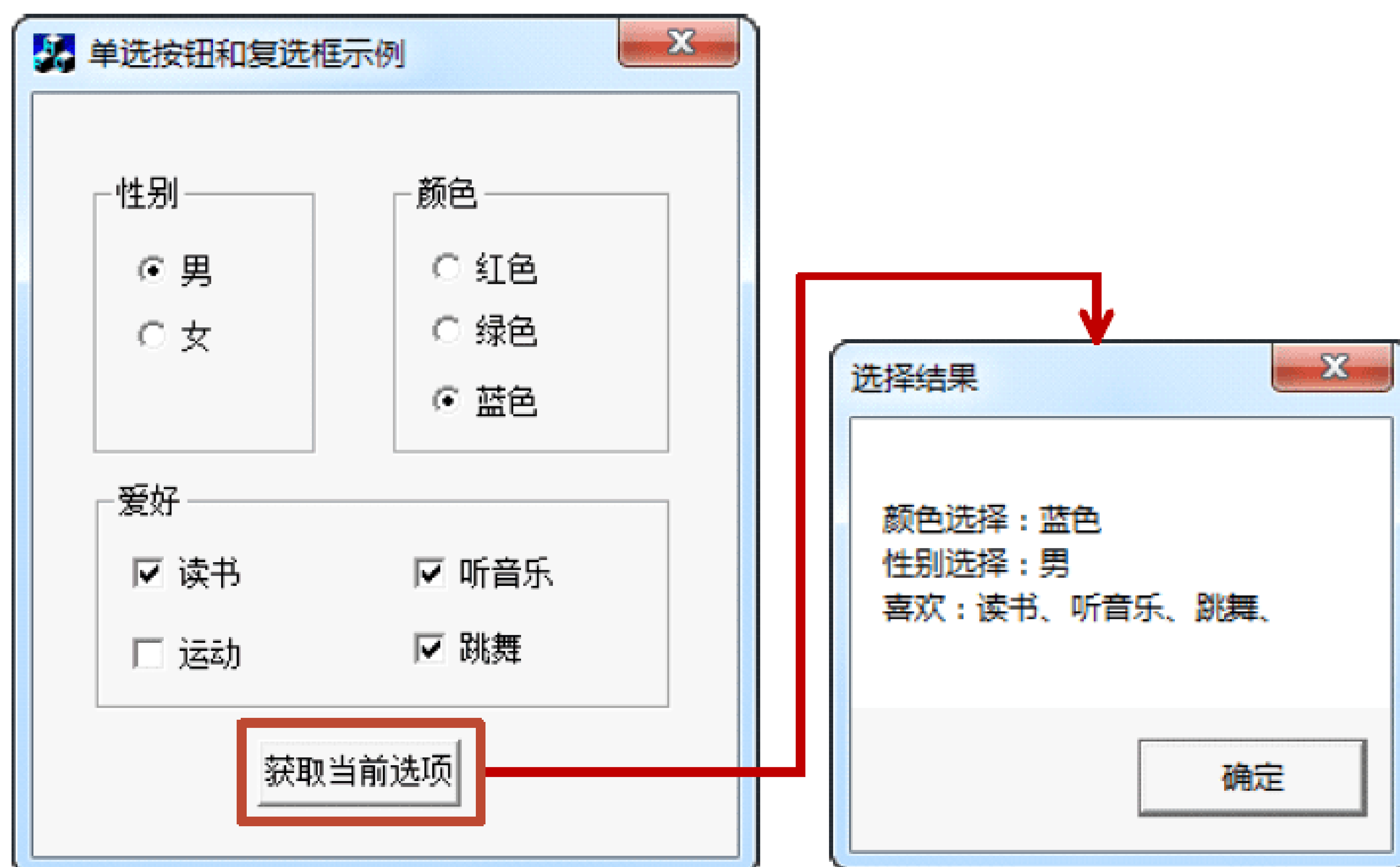


图 7-12　单选按钮控件和复选框控件的实例运行结果

7.5　列表框和组合框

列表框控件是用于从已知选项中选择选项的控件。组合框控件是编辑控件和列表框控件的组合，既具有编辑控件的输入文本功能，又具有列表框控件的选项选择功能。本节将分别介绍列表框和组合框控件的使用。

7.5.1　创建列表框

列表框显示数据项的列表，如文件名等，用户可以浏览和选择数据项。在单选列表框控件中，用户一次只能选择一项。在多选列表框中，可以选择选项范围。当用户选择一项时，会高亮显示，并且列表框控件发送通知消息给父对话框。

创建列表框控件的方法与 7.1.2 小节中介绍的方法类似。只是在创建时，在控件工具栏中选择其中的列表框控件。

7.5.2　列表框类 CListBox

MFC 中使用 CListBox 类提供 Windows 列表框功能。在对话框类中，使用列表框控件时需要在对话框类中声明一个列表框变量，具体方法在 7.1.5 小节中介绍过。使用向导添加了控件对象变量后，向导会在对话框类的 DoDataExchage()函数中使用 DDX_Control()函数将控件与成员变量连接起来。代码如下：

```
01  //对话框数据交换函数
02  void CLBAndCBSampleDlg::DoDataExchange(CDataExchange* pDX)
03  {
04      CDialog::DoDataExchange(pDX);          //调用基类的数据交换函数
05      //列表框对应的变量
06      DDX_Control(pDX, IDC_LIST_TEST, m_listTest);
07      //组合框对应的变量
08      DDX_Control(pDX, IDC_COMBO_TEST, m_comboTest);
09  }
```

上面代码将列表框控件对象变量 m_listTest 与列表框控件 IDC_LIST_TEST 关联起来。使用此对象变量就可以调用 CListBox 类的成员函数。表 7-4 列出了 CListBox 类常用的成员函数。

表 7-4　列表框控件的主要成员函数

成 员 函 数	功　　能
GetCount()	返回列表框控件中的字符串选项的个数
GetHorizontalExtent()	设置可以水平滚动的宽度像素数
SetHorizontalExtent()	设置可以垂直滚动的宽度像素数
GetTopIndex()	返回列表框控件中的第一个可视字符串的索引
SetTopIndex()	设置列表框控件中的第一个可视字符串的索引
GetItemData()	返回与列表框控件项相关的 32 位的值
GetItemDataPtr()	返回与列表框控件项的指针
SetItemData()	设置与列表框控件项相关的 32 位的值
SetItemDataPtr()	设置与列表框控件项的指针
SetItemHeight()	设置列表框控件中项的高度
GetItemHeight()	获取列表框控件中项的高度
GetText()	复制列表框控件项的内容到缓冲区中
GetTextLen()	返回列表框控件项的内容的长度
SetColumnWidth()	设置多列列表框控件的列宽
GetCurSel()	返回列表框控件中当前选择的字符串的索引

续表

成 员 函 数	功　　能
SetCurSel()	选择列表框控件字符串
SetSel()	选择或取消选择多列列表框控件中的列表项
GetSelCount()	返回列表框控件中当前选择的字符串的个数
GetSelItems()	返回列表框控件中当前选择的字符串
AddString()	向列表框中增加字符串
DeleteString()	从列表框中删除字符串
InsertString()	向列表框控件指定位置中增加字符串
ResetContent()	清除列表框控件中的所有选项
Dir()	从当前路径增加文件名到列表框控件
FindString()	从列表框控件中查找字符串
SelectString()	在单选列表框控件中查找和选择字符串

7.5.3　列表框消息

列表框控件常用的消息有以下几个。

- ON_LBN_DBLCLK 消息：当用户双击列表框控件中的字符串时，列表框控件发送此消息给父对话框。只有具有 LBS_NOTIFY 样式的列表框控件才会发送此通知消息。
- ON_LBN_ERRSPACE 消息：当列表框控件不能分配足够的内存处理特殊请求时，发送此消息。
- ON_LBN_KILLFOCUS 消息：当列表框控件失去焦点时，发送给父对话框此消息。
- ON_LBN_SELCANCEL 消息：取消列表框控件的当前选择。只有具有 LBS_NOTIFY 样式的列表框控件才会发送此消息。
- ON_LBN_SELCHANGE 消息：当列表框控件的选择发生变化时，发送给父对话框此消息。如果使用 CListBox::SetCurSel()成员函数改变当前的选择，则列表框控件不会发送选择通知。只有具有 LBS_NOTIFY 样式的列表框控件才会发送此消息。对于多选列表框控件，当用户按下箭头键时，即使选择没有发生变化，也会发送此消息。
- ON_LBN_SETFOCUS 消息：列表框控件接收到输入焦点时，发送给父对话框此消息。
- ON_WM_CHARTOITEM 消息：没有字符串的自绘列表框控件会接收 WM_CHAR 消息。
- ON_WM_VKEYTOITEM 消息：具有 LBS_WANTKEYBOARDINPUT 样式的列表框控件会接收 WM_ KEYDOWN 消息。

7.5.4　列表框实例

下面的代码演示了列表框控件的使用示例。

```
01  //初始化列表框中的数据
02  void CLBAndCBSampleDlg::InitListBoxData()
03  {
04      //列表框中的数据数组
05      CString items[3]= {"北京", "上海", "广州"};
06      for (int i =0;i < 3;i++)
07      {
08          //依次将数据数组中的数据添加到列表框控件
09          m_listTest.AddString(items[i]);
10      }
11  }
12  //改变列表框选择消息处理函数
13  void CLBAndCBSampleDlg::OnSelchangeListTest()
14  {
15      int index = m_listTest.GetCurSel();     //获取当前选择的列表项
16      CString result;                         //定义显示结果变量
17      m_listTest.GetText(index, result);      //获取选择的列表项到变量中
18      MessageBox(result, "当前列表框选择的内容");//显示获取的列表项内容
19  }
```

在上面的代码中，InitListBoxData()函数用于初始化列表框控件中的数据内容。OnSelchangeListTest()函数是列表框控件选择的项发生变化时调用的处理函数，它调用GetCurSel()函数获取当前选择的项，并以消息框的形式显示获取的内容。

7.5.5　创建组合框

组合框控件是列表框与静态控件或编辑控件的组合。控件的列表框部分，可以一直显示，也可以只有当用户单击控件旁的下拉箭头时才显示。列表框中当前选择的数据项，会显示在静态控件或编辑控件中。另外，如果组合框具有下拉列表样式，用户可以输入列表中的其中一项的初始字母，如果存在，则会高亮显示使用这个初始字母的下一项。表 7-5 中列出了组合框控件支持的 3 种样式。

表 7-5　组合框控件的 3 种样式

样　式	何时显示其中的列表框部分	静态控件还是编辑控件
简单样式	总是显示列表框部分	编辑控件
下拉样式	当单击下拉箭头时	编辑控件
下拉列表样式	当单击下拉箭头时	静态控件

创建组合框控件的方法与 7.1.2 小节中介绍的方法类似。只是在创建时，在控件工具栏中选择其中的组合框控件。要设置组合框控件的类型，则右击该组合框控件，在弹出的快捷菜单中选择“属性”命令，打开“属性”对话框，如图 7-13 所示。选择其中的 Type 列表项，下拉列表框中选择组合框使用的样式。

7.5.6　组合框类 CComboBox

CComboBox 类实现 Windows 组合框控件的功能。列表框控件具有 CListBox 类的部分函数，编辑控件具有 CEdit 类的部分函数。除了这些函数，还具有与其自身特点相关的成

员函数，如表 7-6 所示。

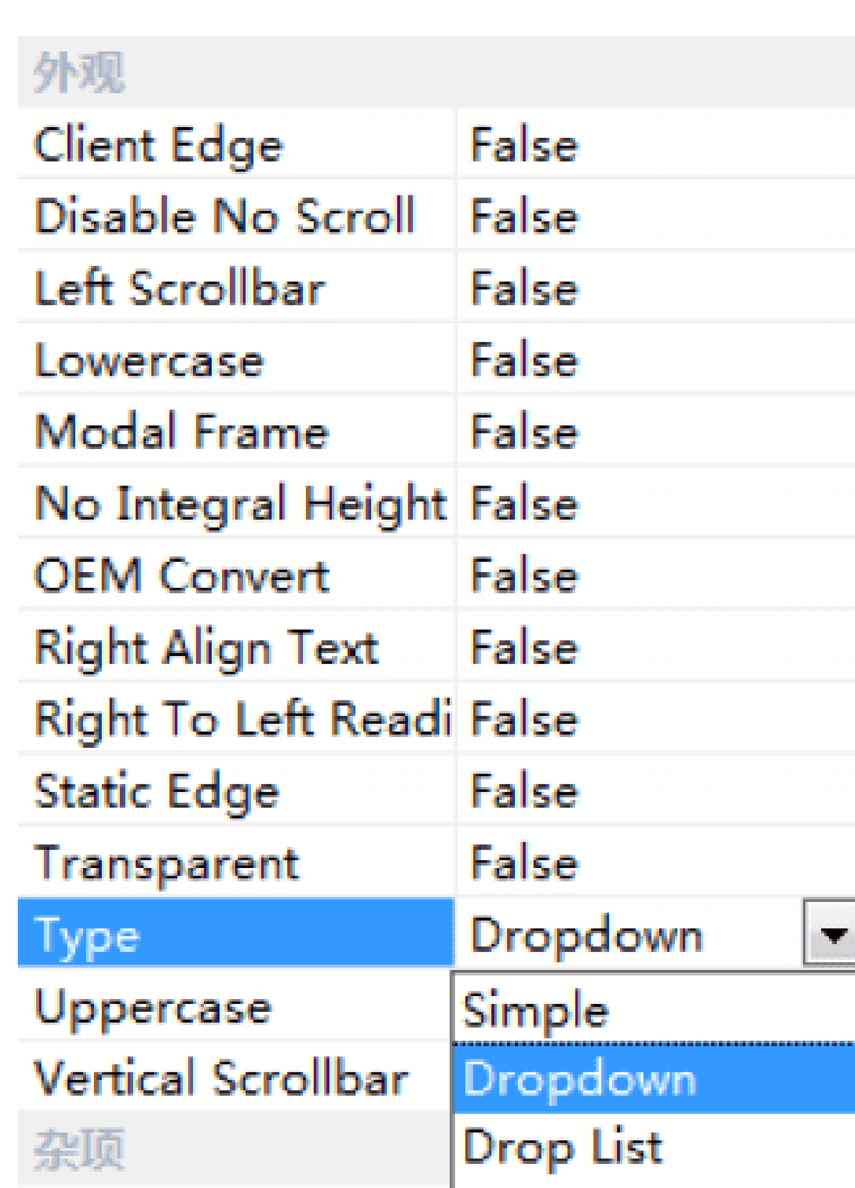


图 7-13　选择组合框样式

表 7-6　CComboBox类的成员函数

成 员 函 数	功　　能
SetDroppedWidth()	设置组合框控件中下拉列表框部分允许的最小宽度
GetDroppedWidth()	获取组合框控件中下拉列表框部分允许的最小宽度
ShowDropDown()	显示或隐藏组合框控件的列表框部分
GetDroppedControlRect()	组合框控件的列表框部分可视的屏幕区域
GetDroppedState()	设定组合框控件的列表框部分是否可见

7.5.7　组合框消息

组合框控件除了可以处理 CWnd 的消息外，还可以处理下面的消息。

- ON_CBN_CLOSEUP 消息：当组合框控件不是 CBS_SIMPLE 样式且组合框控件的列表框部分关闭时，发送此消息给父窗体。
- ON_CBN_DBLCLK 消息：当用户双击组合框控件的列表框部分中的字符串时，发送此消息给父窗体。此消息仅对使用 CBS_SIMPLE 样式的组合框控件有效。
- ON_CBN_DROPDOWN 消息：当用户要下拉组合框控件的列表框部分时，发送此消息给父窗体。此消息仅对具有 CBS_DROPDOWN 样式或 CBS_DROPDOWNLIST 样式的组合框控件有效。
- ON_CBN_EDITCHANGE 消息：当用户修改组合框控件的编辑控件中的内容时，触发此消息。与 CBN_EDITUPDATE 消息不同，消息在 Windows 更新完屏幕后才会发送此消息。对于 CBS_DROPDOWNLIST 样式的组合框控件，此消息无效。
- ON_CBN_EDITUPDATE 消息：当组合框控件的编辑控件部分要显示修改的文本时，发送此消息。此通知消息在控件格式化完文本内容但是显示文本前，发送消息。对于具有样式 CBS_DROPDOWNLIST 的组合框控件无效。
- ON_CBN_ERRSPACE 消息：当组合框控件不能分配足够的内存处理特殊请求时，

发送此消息。

- ON_CBN_SELENDCANCEL 消息：表示取消用户的选择。当用户单击某一项，然后单击其他窗体或控件隐藏组合框控件的列表框部分时，在发送 CBN_CLOSEUP 消息之前发送此消息，用于表示会忽略用户的选择。当组合框控件是 CBS_SIMPLE 样式时，即使不发送 CBN_CLOSEUP 通知消息，也会发送 CBN_SELENDCANCEL 或 CBN_SELENDOK 通知消息。
- ON_CBN_SELENDOK 消息：用户选择其中一项，并且按下 Enter 键或单击下拉箭头隐藏组合框控件的列表框控件时，发送此消息。
- ON_CBN_KILLFOCUS 消息：组合框控件失去焦点时，触发此消息。
- ON_CBN_SELCHANGE 消息：当组合框控件的选择发生变化时，触发此消息。在处理此消息时，如果要获取组合框控件中的编辑控件中的文本内容，则 X 需要通过 GetLBText()函数获取，而不能使用 GetWindowText()函数。
- ON_CBN_SETFOCUS 消息：当组合框控件得到焦点时，触发此消息。

7.5.8　组合框实例

下面的代码演示了组合框控件的使用示例。

```
01  void CLBAndCBSampleDlg::InitComboBoxData()
02  {
03      //组合框中的数据数组
04      CString items[5]= {"汉族", "回族", "满族", "白族", "其他"};
05      for (int i = 0;i < 5;i++)
06      {
07          //依次将数据数组中的数据添加到组合框控件
08          m_comboTest.AddString(items[i]);
09      }
10  }
11  //改变组合框选择消息处理函数
12  void CLBAndCBSampleDlg::OnSelchangeComboTest()
13  {
14      CString result;                              //定义显示结果变量
15      m_comboTest.GetWindowText(result);           //获取组合框内容到变量中
16      MessageBox(result, "当前组合框选择的内容");  //显示组合框内容
17  }
```

在上面的代码中，InitComboBoxData()函数用于初始化组合框控件中的数据内容。OnSelchangeComboTest()函数是组合框控件选择的项发生变化时调用的处理函数，它调用 GetWindowText()函数获取组合框中当前的数据，并以消息框的形式提示给用户。

7.6　微调控件、滑块控件和进度条控件

Windows 中提供了 3 种带有刻度功能的控件，分别是微调控件、滑块控件和进度条控件。微调控件用于控制连续的整数值调整。滑块控件通过拖放滑块控件表示的进度。进度条控件以动态滚动的方式显示当前程序的进度。本节将分别介绍这 3 种控件的使用。

7.6.1　微调控件的创建和使用

微调控件也称为上下控件，提供一组箭头，单击箭头可以调整其值。此值称为当前位置，这个位置值必须在微调控件的范围内。当用户单击向上箭头时，位置值向最大值移动；当用户单击向下箭头时，位置值向最小值移动。

创建微调控件的方法与 7.1.2 小节中介绍的方法类似。只是在创建时，在控件工具栏中选择其中的微调控件。要使编辑框与微调控件的取值一致，需要执行如下操作。

（1)将编辑框控件的 Tab 顺序值与微调控件的 Tab 顺序键的值相邻，并且编辑框控件的 Tab 键顺序值更大。

（2）取消微调控件的 Tab 属性。

（3）设置微调控件的属性，如图 7-14 所示。

MFC 中使用 CSpinButtonCtrl 类实现微调控件的功能。微调控件的默认范围是 0～100。因此，当按下向上箭头时，减少位置值；当按下向下箭头时，增加位置值。但是可以使用 CSpinButtonCtrl::SetRange()成员函数调整范围值。

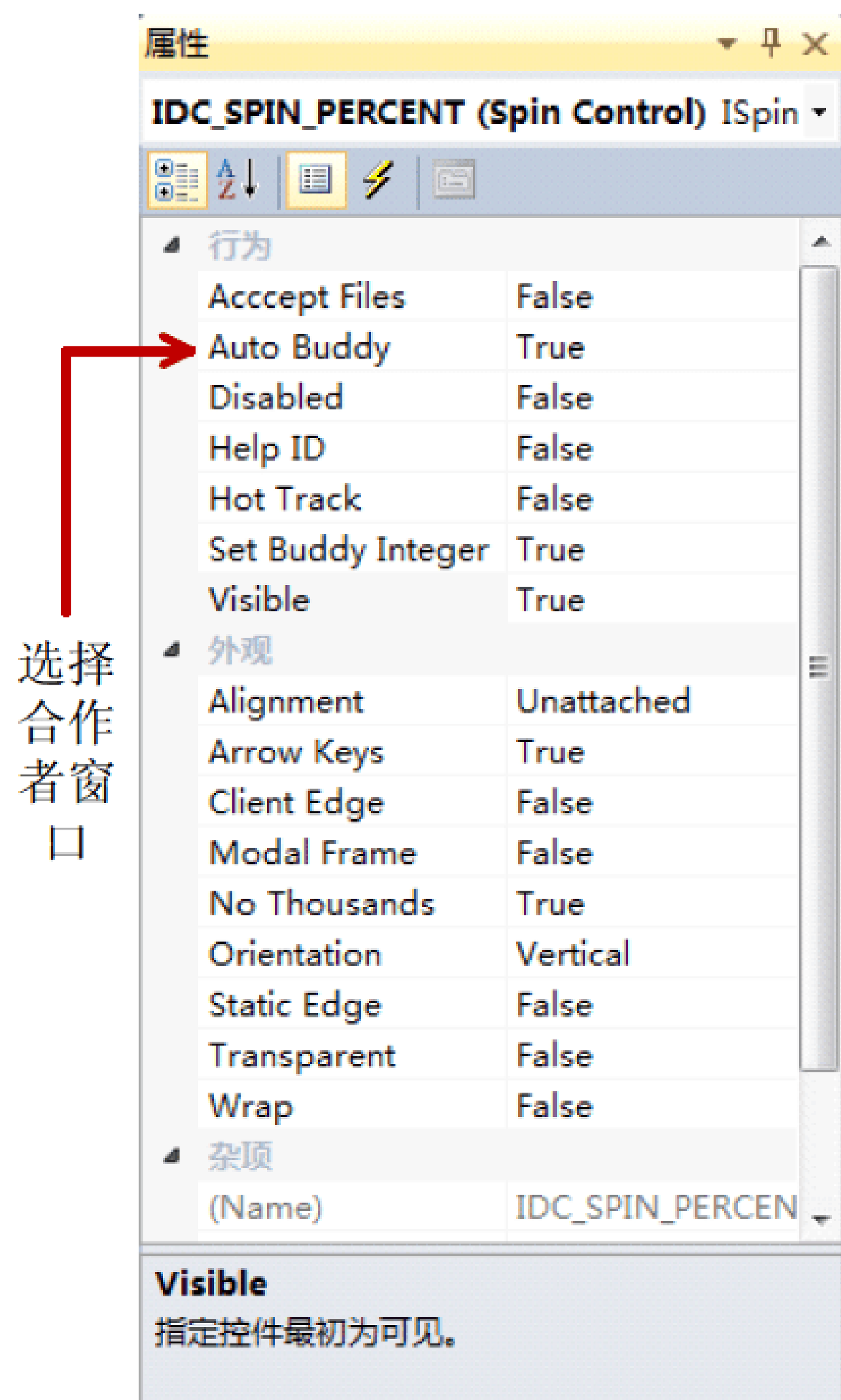


图 7-14　微调控件的属性设置

7.6.2　创建和使用滑块控件

滑块控件又称为跟踪条，包含滑块和可选的标记线。当用户移动滑块时，滑块控件会发送改变取值的通知消息给父对话框。如在控制面板中设置键盘速度时，使用的就是滑块控件。当滑块控件移动时，按照创建时指定的增量移动滑块位置。如果指定滑块控件有 10 个范围值，则滑块控件只有 11 个位置：一个是滑块最左边的位置，还有范围内每个增量位置，这些增量位置使用标记定义。

创建滑块控件的方法与 7.1.2 小节中介绍的方法类似。只是在创建时，在控件工具栏中选择其中的滑块控件。滑块控件支持一些界面的外观设置，如图 7-15 所示。

在图 7-15 所示的对话框中，加入了两个滑块。一个是垂直滑块，一个是水平滑块。“属性”对话框中的 Orientation 列表项，可以设定滑块是水平的还是滚动的。

其中，Point 列表项可以设定滑块的箭头的方向，包括双向的、向左或向上的、向右或向下的 3 种方向。图 7-15 所示中的水平滑块的箭头方向是双向的，垂直滑块的箭头方向是向左的；Tick marks 列表项表示是否具有滑块的标记；Auto ticks 列表项表示是否在滑块上自动添加等分标记；Enable selection 列表项表示是否带有选择部分；Border 列表项表示滚动条是否具有边框。在图 7-15 中，垂直滚动条有边框，水平滚动条没有边框。读者可以根据这些属性设计需要的滑块界面。

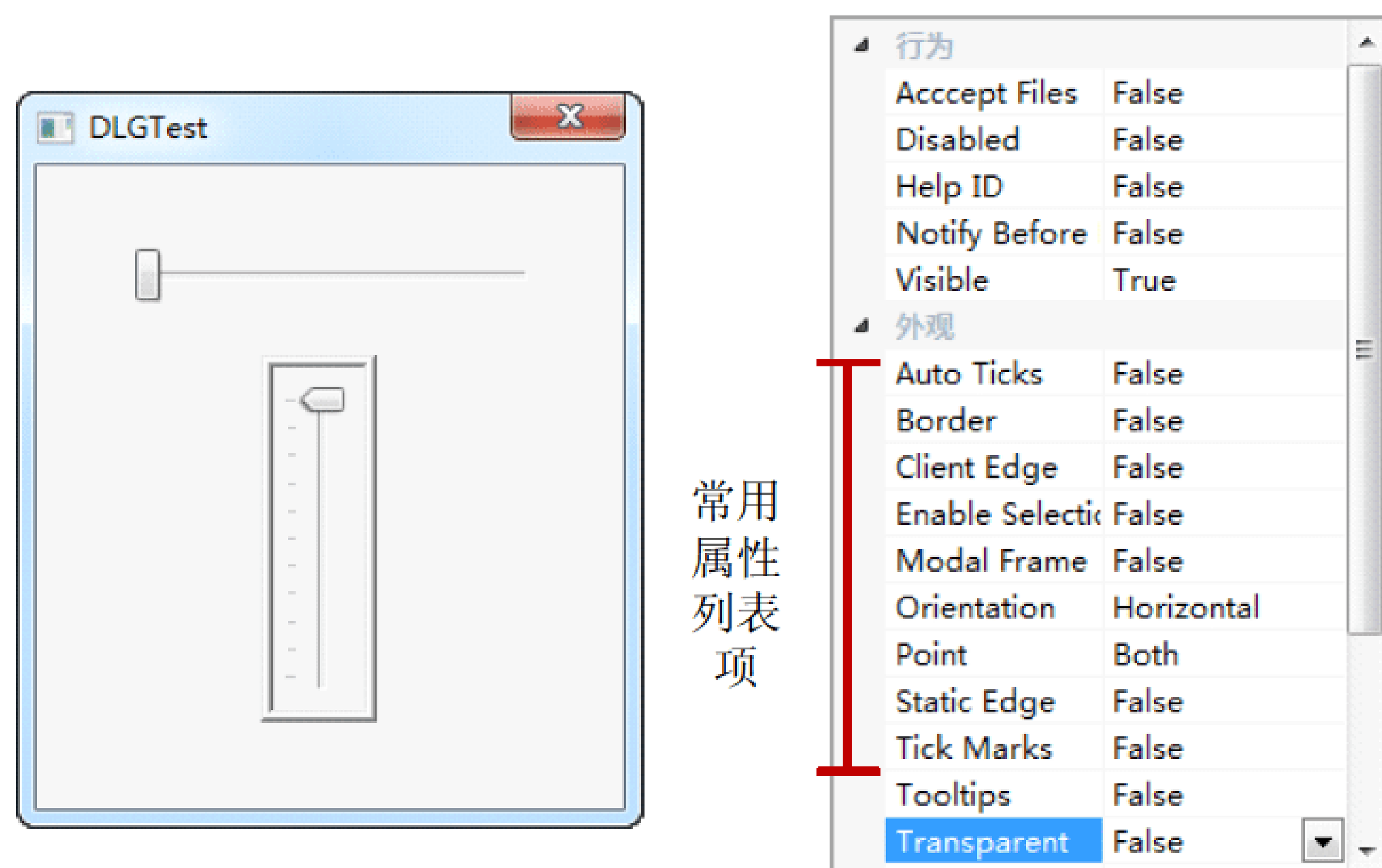


图 7-15　滑块控件的属性

7.6.3　创建和使用进度条控件

创建进度条控件的方法与 7.1.2 小节中介绍的方法类似。只是在创建时，在控件工具栏中选择其中的进度条控件。MFC 中使用 CProgressCtrl 类完成进度条的功能。它最重要的方法有下面 4 个。

- SetRange()成员函数：可以设置进度条控件的范围值。
- SetStep()成员函数：可以设置进度条控件的增量间隔值。
- SetPos()成员函数：可以设置进度条控件的当前位置值。
- GetPos()成员函数：可以获取进度条控件的当前位置值。

7.6.4　编程实例

下面的示例演示了这 3 种进度控件的使用方法。

```
01  //初始化微调控件参数
02  void CSpinSliderProgressSampleDlg::InitSpinData()
03  {
04      m_spinPercent.SetRange( 0, 100 );  //设置微调控件支持的范围为 0～100
05      m_spinPercent.SetBase( 10 );       //设置微调控件的步长为 10
06      m spinPercent.SetPos( 0 );         //设置微调控件当前值为 0
07      return ;                           //返回
08  }
09  //初始化滑块控件
10  void  CSpinSliderProgressSampleDlg::InitSliderData()
11  {
12      //设置滑块控件的范围为 0～100
13      m_sliderPercent.SetRange(0, 100, TRUE );
14      m_sliderPercent.SetTic(10);        //设置滑块控件的单位滑动长度为 10
15      m_sliderPercent.SetPos(10);        //设置滑块控件的当前位置为 0
16      return ;                           //返回
17  }
```

```
18  //初始化进度条数据
19  void  CSpinSliderProgressSampleDlg::InitProgressData()
20  {
21      m progressPercent.SetRange(0, 100);//设置进度条控件的范围为 0～100
22      m_progressPercent.SetStep(10);     //设置进度条控件的步长为 10
23      m_progressPercent.SetPos(0);       //设置进度条控件的当前位置为 0
24      return ;
25  }
26  //释放滑块控件滑动后的处理函数
27  void CSpinSliderProgressSampleDlg::OnReleasedcaptureSliderPercent(
28      NMHDR* pNMHDR, LRESULT* pResult)
29  {
30      CString text;                      //定义字符串变量
31      //获取滑块控件当前的位置
32      text.Format("%d", m_sliderPercent.GetPos());
33      //在编辑控件中显示滑块控件当前的选择值
34      m editPercent.SetWindowText(text);
35      *pResult = 0;                      //pResult 置为 0
36  }
37  //定时器按钮处理函数
38  void CSpinSliderProgressSampleDlg::OnButtonTimer()
39  {
40      m progressPercent.SetPos(0);       //设置滑块控件的位置为 0
41      SetTimer(100, 100, NULL);          //启动定时器
42  }
43  //定时器处理函数
44  void CSpinSliderProgressSampleDlg::OnTimer(UINT nIDEvent)
45  {
46      //如果定时器是滑块滑动定时器
47      if (nIDEvent == 100)
48      {
49          int pos = m_progressPercent.GetPos();//获取进度条位置
50          //如果进度条位置在有效范围内，设置进度条控件的位置递增 1
51          if (pos < 100)
52              m progressPercent.SetPos(m progressPercent.GetPos()+1);
53          else
54              KillTimer(100); //否则关闭定时器
55      }
56      CDialog::OnTimer(nIDEvent);
57  }
```

这个例子中，InitSpinData()函数、InitSliderData()函数和 InitProgressData()函数分别用于初始化微调控件、滑块控件和进度条控件的范围以及增量值。OnReleasedcapture-SliderPercent()函数是当滑块控件的滑块值发生变化时的处理函数，其会更新编辑框中的值。OnButtonTimer()函数使单击测试滚动条的按钮时启动定时器，定时器处理函数会向前滚动一格位置。

7.7　列表视图控件和树形视图控件

列表视图控件扩展了列表框控件的功能，用于显示并列级别的数据信息。树形视图控

件用于显示层次结构的数据项。这两种工具在 Windows 程序中经常用到，包括 Windows 的核心工具——资源管理器中都使用了这两种控件。本节将介绍有关这两种控件的使用方法。

7.7.1　创建列表视图控件

列表视图控件显示包含图标和标签的项的集合。除了图标和标签，每项可以在图标和标签的右边显示信息。最常见的列表视图控件的应用就是 Windows 操作系统的资源管理器。

列表视图控件支持以下 4 种显示方式，即视图样式。

- ❑ Icon 视图：即图标视图。此种视图样式下，每个数据项显示时，在完整尺寸的图标下显示标签。用户可以拖动数据项到列表视图对话框中的任何位置。
- ❑ Small icon 视图：即小图标视图。此种视图样式下，每个数据项显示时，使用小图标（16×16 像素）显示，并在右边显示标签。用户可以拖动数据项到列表视图对话框中的任何位置。
- ❑ List 视图：即列表视图。此种视图样式下，当每个数据项显示时，使用小图标显示，并在右边显示标签。数据项是以列的方式排列，而且此种视图方式下，用户不可以拖动数据项到列表视图对话框中的其他位置。
- ❑ Report 视图（报表视图）：此种视图样式下，每个数据项显示一行，并且除了名称外，其他信息在名称的右边列中列出。最左边的列包含小图标和标签，后续的列由应用程序指定子项。它内置了一个标题控件（CHeaderCtrl）以实现这些列。

图 7-16 中分别显示了 4 种列表视图的样式，依次为：Icon、Small Icon、List 和 Report。

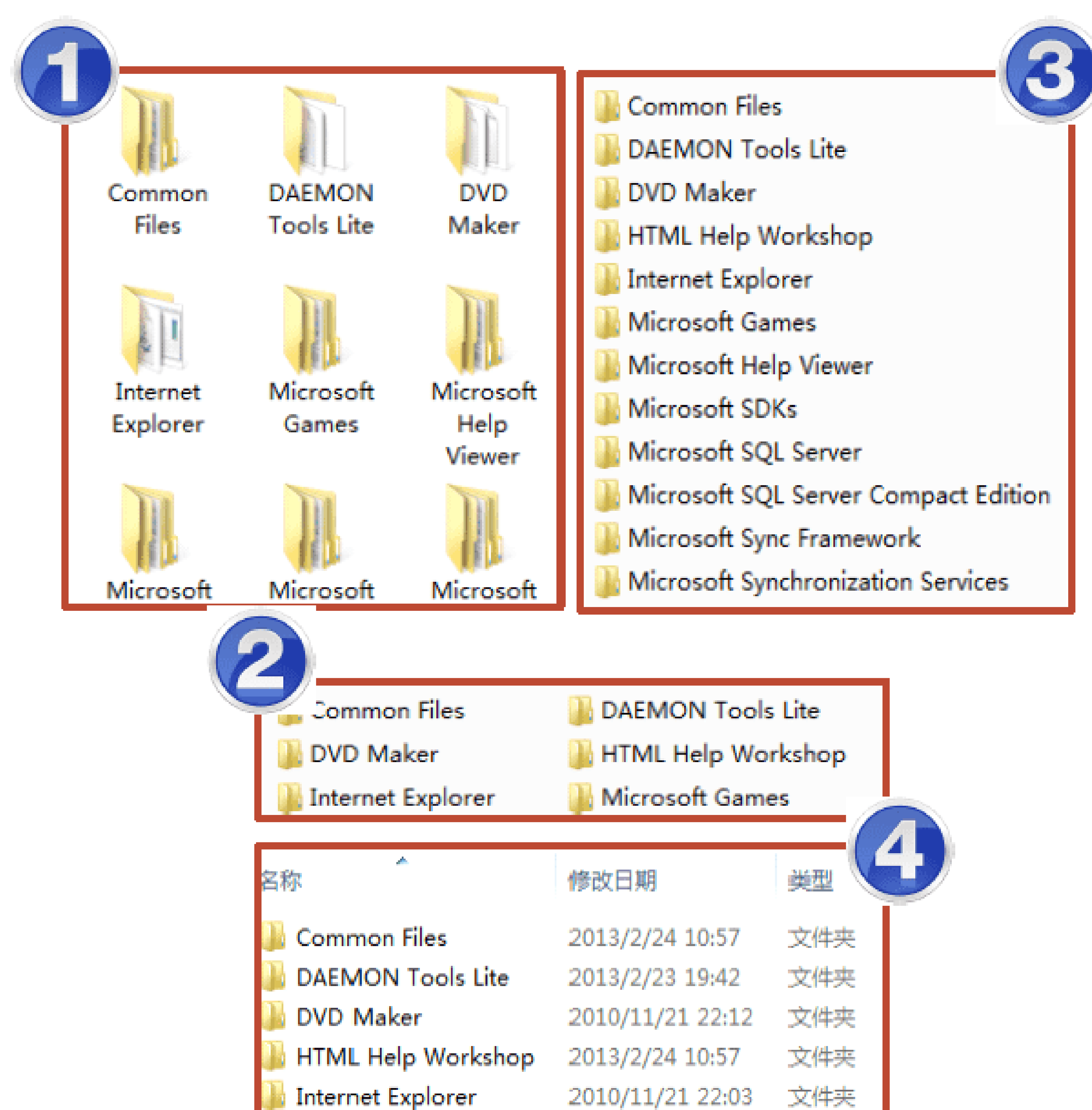


图 7-16　列表视图样式

创建列表视图控件的方法与 7.1.2 小节中介绍的方法类似。只是在创建时，在控件工

具栏中选择其中的列表视图控件。

7.7.2　列表视图控件类 CListCtrl

CListCtrl 类封装了列表视图控件的功能。列表视图控件中的每项都包含一个图标、一个标签、当前状态和应用程序定义的数据，并且每项都可以具有与其相连的子项。这些子项可以在列表视图中的其他列中显示，而且列表视图控中的所有项都必须具有相同的子项数目。CListCtrl 类常用的方法有 3 个。

- ❑ CListCtrl::GetItem()方法：可以获取指定索引处的项的数据。
- ❑ CListCtrl::InsertItem()方法：可以在指定索引处添加新项。
- ❑ CListCtrl::FindItem()方法：可以查找指定项。

列表视图控件中使用 CImageList 类实现图像列表。每个控件应该具有 4 种不同的图像列表，如下所述。

- ❑ 大图标：在显示完整大小的图标时使用。
- ❑ 小图标：在小图标样式、列表样式和报表样式时使用。
- ❑ 用户自定义状态：包含状态图片，用于显示用户定义的状态。
- ❑ 标题头项：显示在标题头项中的图片。

7.7.3　列表视图控件的通知消息

当用户执行单击列标题、拖动图标以及编辑标签等操作时，CListCtrl 会向父窗体发送通知消息。如果要响应用户的操作，则应该处理这些函数。如果当用户单击列标题时，要对项按照此列内容进行排序，则应该在处理函数中处理。

在视图类或对话框类中处理 WM_NOTIFY 消息对应的处理函数 OnChildNotify()时，可加入 switch 语句处理不同的消息。

7.7.4　创建树形视图控件

树形视图控件是显示层次结构的项，如显示磁盘文件中的逻辑层次。每项包含标签和可选的位图图像，而且每项也可以包含与其相连的子项。单击其中的一项，可以展开和收缩预期相连的子项。创建树形视图控件的方法与 7.1.2 小节中介绍的方法类似，只是在创建时，在控件工具栏中选择其中的树形视图控件。

7.7.5　树形视图控件类 CTreeCtrl

CTreeCtrl 类提供树形视图控件的功能，它是实现层次项的窗口，如磁盘上的文件项。每项包含一个标签和一个可选的位图图片，并且每项都可以包含与之相连的子项。通过单击每项可以展开和收缩与之关联的子项的列表。如表 7-7 列出了 CTreeCtrl 类的成员函数。

表 7-7　CTreeCtrl类的成员函数

成 员 函 数	功　　能
GetCount()	返回与视图控件相连的项的数目
GetNextItem()	返回视图控件中下一个符合要求的项
ItemHasChildren()	返回指定的项是否具有子项
GetChildItem()	返回执行项的子项
GetNextSiblingItem()	返回下一个兄弟项
GetPrevSiblingItem()	返回上一个兄弟项
GetParentItem()	返回指定项的父项
GetFirstVisibleItem()	返回指定项的第一个可视项
GetNextVisibleItem()	返回指定项的下一个可视项
GetPrevVisibleItem()	返回指定项的上一个可视项
GetSelectedItem()	返回当前选择的项
GetRootItem()	返回根项
GetItem()	返回视图项的属性
SetItem()	设置视图项的属性
GetItemImage()	返回与指定项相关的图像
SetItemImage()	设置指定项的图像
GetItemText()	返回指定项的文本
SetItemText()	设置指定项的文本
InsertItem()	向控件中插入新项
DeleteItem()	从控件中删除项
DeleteAllItems()	删除所有项
Expand()	展开或收缩指定项下的子项

7.7.6　树形视图控件的消息

CTreeCtrl 类发送的 WM_NOTIFY 消息，如表 7-8 所示。

表 7-8　CTreeCtrl类的WM_NOTIFY 消息

消　　息	含　　义
TVN_BEGINDRAG	开始拖动操作时的通知消息
TVN_BEGINLABELEDIT	开始编辑标签内容时的通知消息
TVN_BEGINRDRAG	使用右键开始拖动操作时的通知消息
TVN_DELETEITEM	删除指定项时的通知消息
TVN_ENDLABELEDIT	结束编辑标签内容时的通知消息
TVN_GETDISPINFO	树形视图控件请求显示项时的请求信息
TVN_ITEMEXPANDED	展开或收缩项时的通知消息
TVN_ITEMEXPANDING	要展开或收缩项时的通知消息
TVN_KEYDOWN	按下键盘时的通知消息
TVN_SELCHANGED	选项变化时的通知消息
TVN_SELCHANGING	要变化选项时的通知消息
TVN_SETDISPINFO	通知要更新项包含的信息

7.7.7　编程实例

下面的代码中，显示了如何将 FTP 服务器中的层次结构在树形视图控件中显示。

```
01  void CFTPSampleView::ShowFiles()                //显示 FTP 站点上的内容
02  {
03      //定义查询 FTP 文件的变量
04      CFtpFileFind ftpFind(m_pFtpConnection);
05      CString strFileName;                        //定义存放 FTP 文件名的变量
06      //定义是否继续查询的变量
07      BOOL bContinue = ftpFind.FindFile(_T("/*"));
08      while (bContinue)                           //依次循环处理 FTP 文件查找
09      {
10          //查找 FTP 文件
11          bContinue = ftpFind.FindNextFile();
12          //获取查找到的 FTP 文件名
13          strFileName = ftpFind.GetFileName();
14          //在列表框控件中显示 FTP 文件名
15          m_fileCtrl.AddString(strFileName);
16      }
17      ftpFind.Close();                            //关闭 FTP 文件查找变量
18  }
```

上面的代码在建立了 FTP 连接后，定义了查询 FTP 文件的变量，循环查找 FTP 文件，并以此将其显示在列表框控件中。

7.8　ActiveX 控件

ActiveX 控件，也称为 OLE 控件，是提供连接点和主机标准接口的 COM 组件。这些标准接口定义可以在控件包含器中处理控件的协议、交换消息和处理事件。ActiveX 控件是 Windows 编程中非常重要的概念。

7.8.1　使用 ActiveX 控件

使用对话框编辑器可以在设计时设置控件属性。如果设置了属性，则资源编辑器会使用指定值初始化控件。而这些属性值仍然可以在编程时修改。步骤如下：

（1）在对话框资源编辑器中右击控件，在弹出的快捷菜单中选择“属性”命令。

（2）选择 All 选项卡，或选择指定选项卡，在属性中输入初始值。

用户使用 ActiveX 控件可以响应 ActiveX 控件的事件，可以使用类向导查看控件中可用的事件，并创建事件处理句柄。

7.8.2　ActiveX 控件的结构

作为 COM 服务器，ActiveX 控件的结构具有下面几部分。

- 属性：ActiveX 控件使用成员变量表示中间状态，成员变量通过 Get()和 Set()访问函数实现后称为属性。idl 文件中使用 propget 标识的每个访问方法都有对应的 Get()函数，idl 文件中使用 propput 或 propputref 标识的每个访问方法都有对应的 Set()函数。可以使用包装类和 OLE/COM 对象查看器确定访问函数的原型。
- 方法：使用公共方法定义的控件行为。包装类提供了访问控件方法的途径。如果使用包装类，通过获取接口的指针访问控件的方法。如 ADO 数据控件中的 Refresh()方法就是公共方法，用于更新获取的行集。有关该方法的使用会在后面的章节中介绍。
- 事件：控件可以使用事件通知宿主程序“有事情发生”。如 Button 按钮控件的 OnClick 事件，当单击按钮控件时，按钮控件会触发 OnClick 事件。如果控件的宿主程序为事件设置了处理函数，则控件此时会调用此函数。
- 类型库：类型库告诉控件包含器，控件支持的属性、方法和事件。控件库可以放在单独的文件中，即扩展名为.tlb 的文件，或者是放在控件内部。控件库还可以包含控件的组件类信息。组件类是使用 GUID 定义的 COM 类，包含控件定义的一个或多个接口，使用 OLE/COM 对象查看器可以查看类型库。

7.8.3　包装类

当使用控件时，类向导会为每个控件的内部组件类生成包装类。这些包装类为组件提供了简单的编程接口。类向导通常会创建多个包装类。读者可以通过查看类名区分控件的包装类。包装类是控件名称前加一个字母 C。也可以从类的基类判断包装类，包装类继承自 CWnd 类，而控件包装类继承自 COleDispatchDriver 类。

通常情况下，只用到与控件相关的包装类。但是有的时候，包装类中的一些 Get()函数也返回其他组件类的指针。如果生成了这些函数，但是返回值对应的包装类没有生成，应用程序不会编译。因此，通常情况下，除非对组件类非常了解，否则最好是在插入控件时，生成其所有的包装类。

使用 CWnd::GetDlgItem()函数也需要包装类，因为返回值必须转换成控件类型。代码如下：

```
CDBList* pDBList = 0;                    //数据库列表变量
//将 IDC_DBLIST 控件映射为变量
pDBList = static_cast<CDBList*>(GetDlgItem(IDC_DBLIST));
```

7.8.4　获取 ActiveX 控件的帮助信息

通常 ActiveX 控件都有自己的属性、方法和事件，其使用方法都不同。这时，就需要开发人员学会使用 ActiveX 控件的帮助信息。其方法有两种：一种方法是查看控件带的帮助文件；另一种是使用 OLE/COM 对象浏览器。查看控件帮助文件的步骤如下：

（1）在对话框编辑器中，右击 ActiveX 控件，在弹出的快捷菜单中选择“属性”命令，弹出控件的“属性”对话框。

（2）在该对话框上单击左上角的第二个帮助按钮，即会弹出对应的帮助文件。

通过查看生成的 idl 文件，读者可以确定控件导出的属性、方法和事件，也可以直接查看方法和访问函数的声明。除了查看 ActiveX 控件的帮助外，还可以使用 OLE/COM 对象查看器获取控件的信息。通过读取控件的类型库，使用户查看控件接口。步骤如下：

（1）选择“工具”|“Visual Studio 命令提示”命令，然后在命令行中输入 oleview 命令，即可弹出 OLE/COM Object Viewer 工具界面。

（2）在左边树形视图中，选择 Object Classes|Grouped by Component Category|Automation Objects 选项。

（3）选择要查看的 ActiveX 控件。在右边面板上会显示一组选项卡，其中 Registry 选项卡中会显示控件实现的接口，如图 7-17 所示。

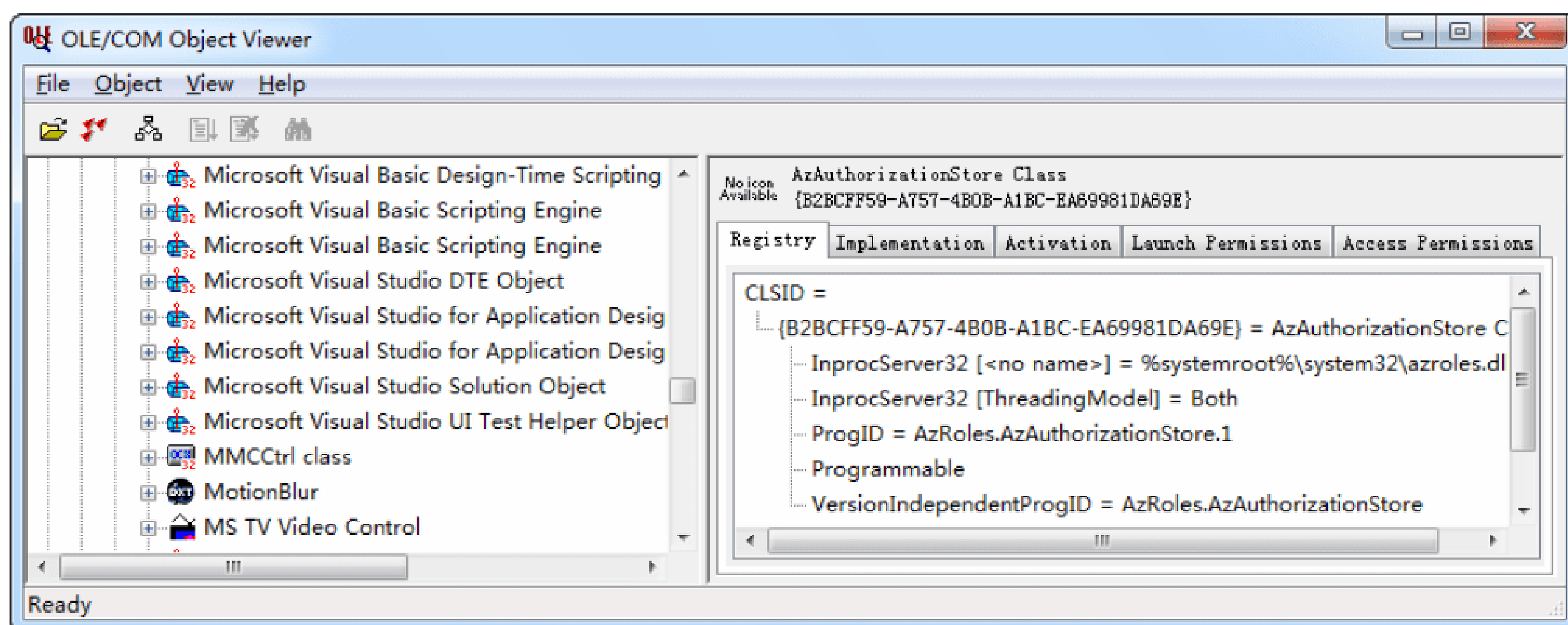


图 7-17　使用 OLE/COM 对象查看器

（4）如果右击左边面板上的控件，在弹出的快捷菜单中选择 View Type Information 命令，则会弹出 ITypeInfo 查看器，显示 ild 或 odl 文件内容。右击左边面板上控件的某个接口，在弹出的快捷菜单中选择 View 命令，则会弹出显示 GUID 的对话框，如果有类型信息，则 View Type Info 按钮会变成可用，单击此按钮，也会弹出 ITypeInfo 查看器，如图 7-18 所示。

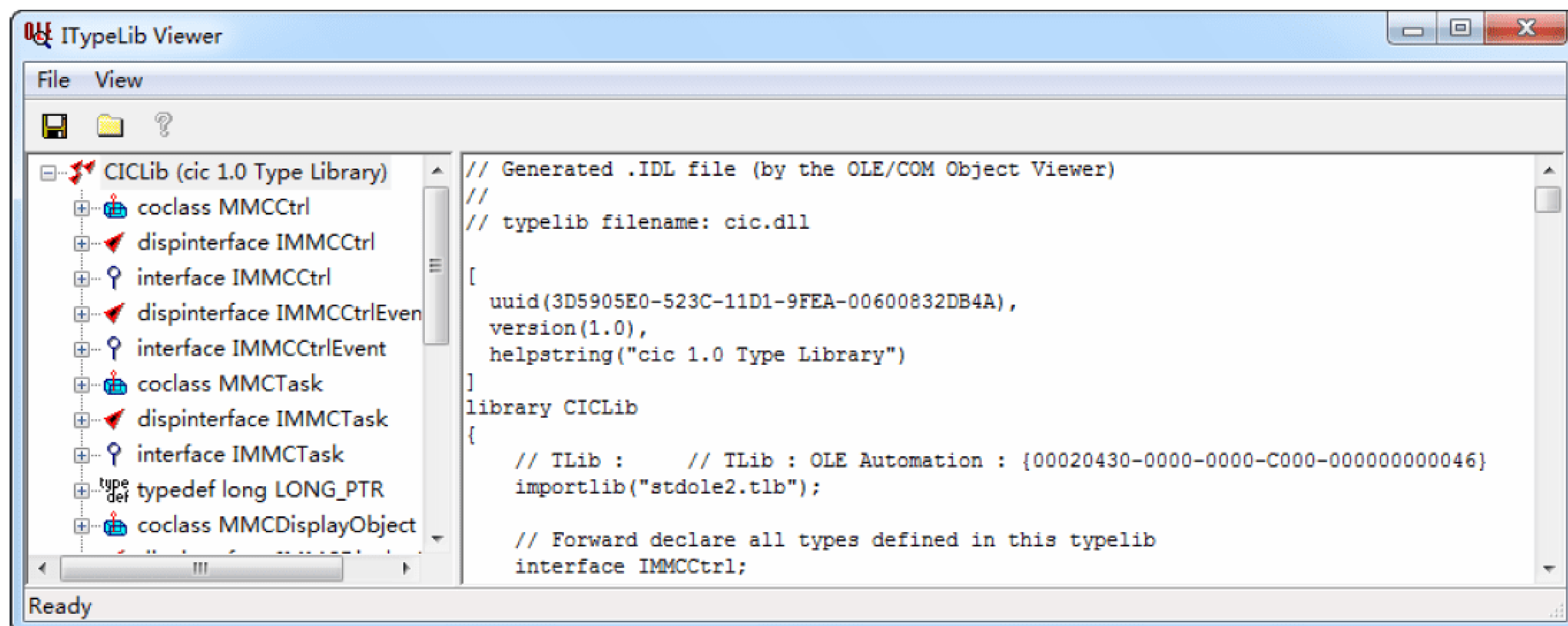


图 7-18　查看类型库

（5）在左边的树形视图中选择要查看的对象接口，则右边面板会显示接口的注册信息。

7.8.5　Visual C++中的控件和组件库

使用 ActiveX 控件有两种方式：一种是从控件和组件库中插入控件到工程中；另一种是在对话框编辑器中插入控件。从控件和组件库中插入 ActiveX 控件的步骤如下：

（1）选择“项目” | “添加类”命令，弹出 “添加类”对话框，如图 7-19 所示。

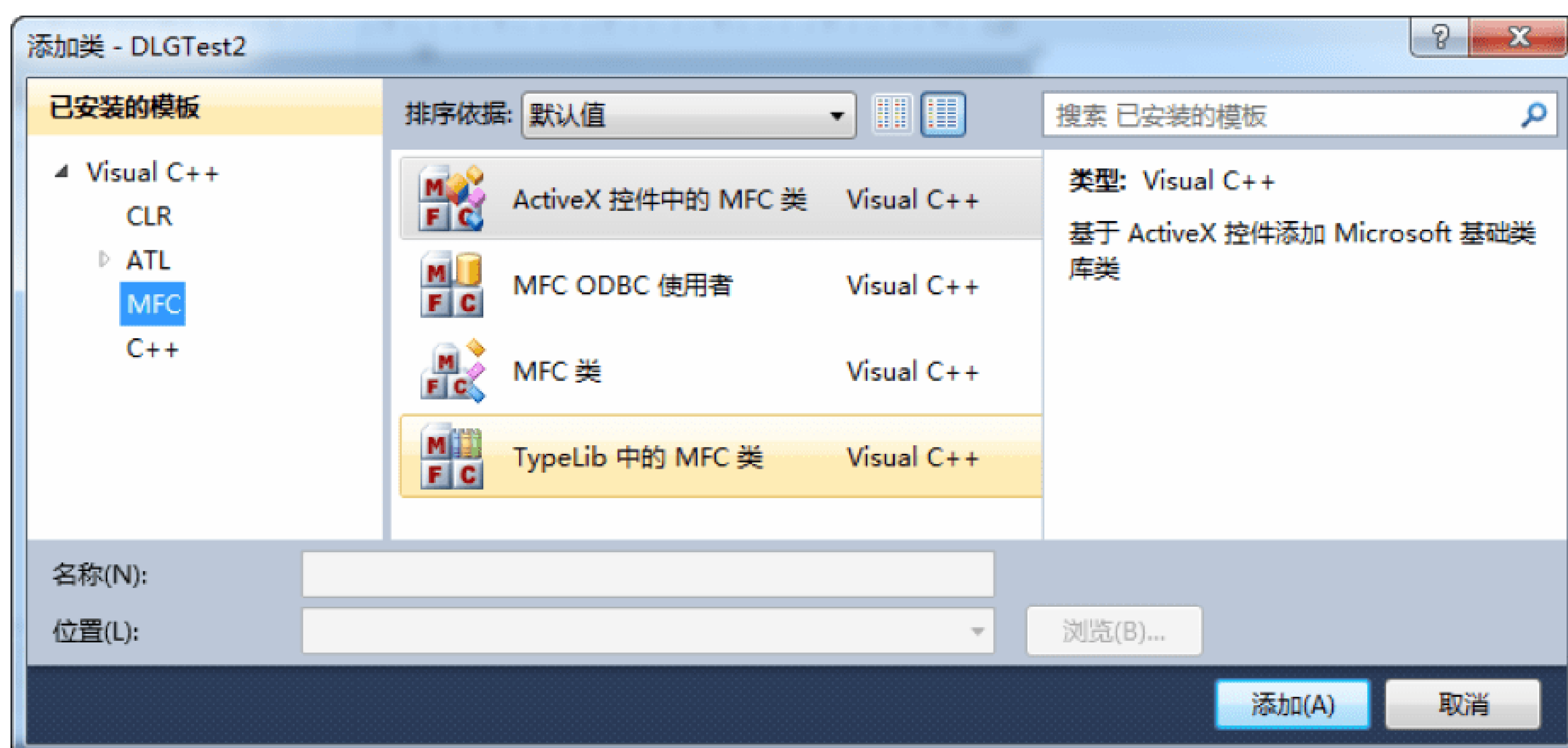


图 7-19 “添加类”对话框

（2）选择“ActiveX 控件中的 MFC 类”，单击“添加”按钮，弹出对话框如图 7-20 所示。

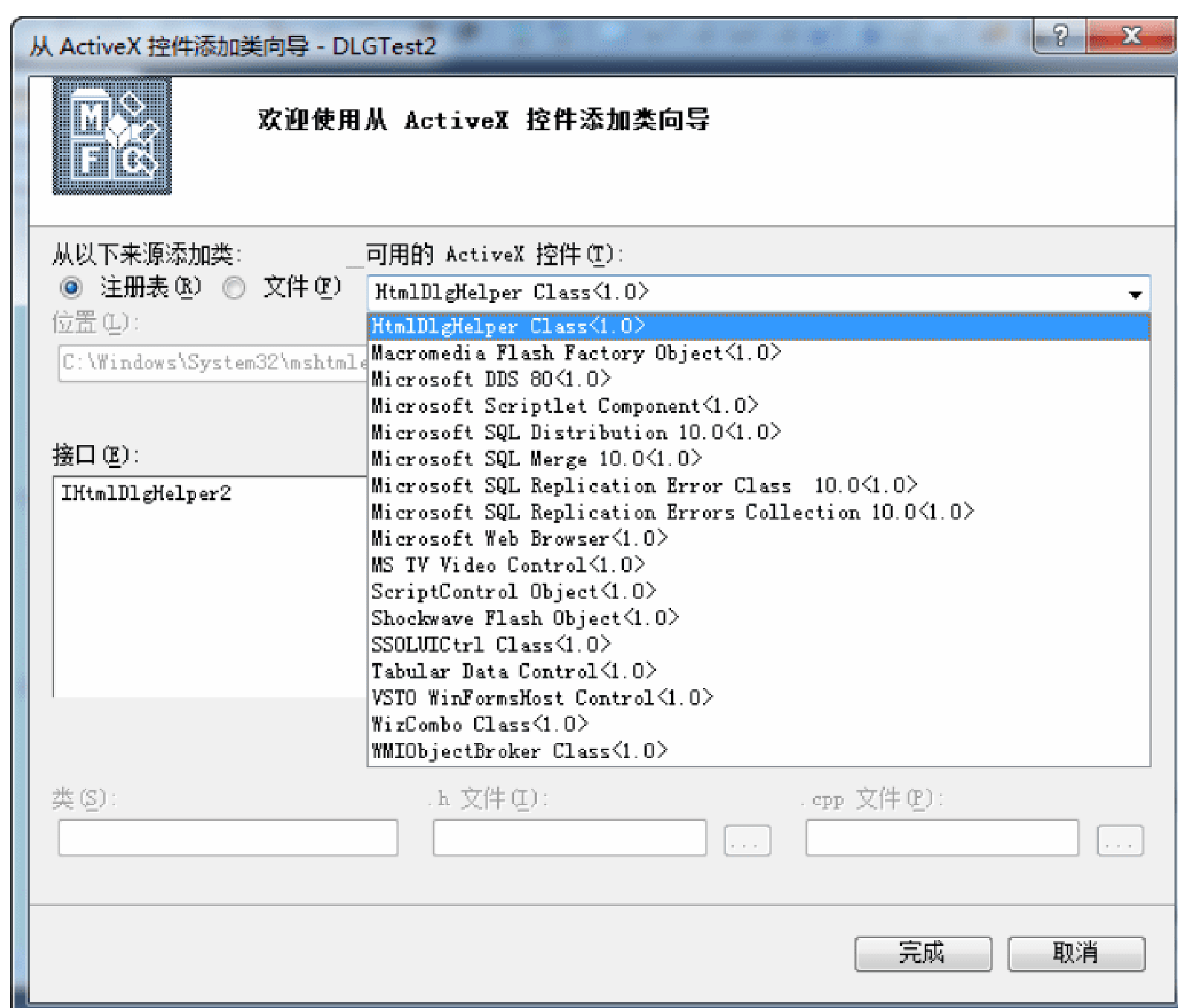


图 7-20　从 ActiveX 控件添加类向导

（3）选择要加入的 ActiveX 控件，如 Microsoft Wed Browser。然后选择 “接口”和“生

成的类”，如图 7-21 所示。最后单击“完成”按钮即可。

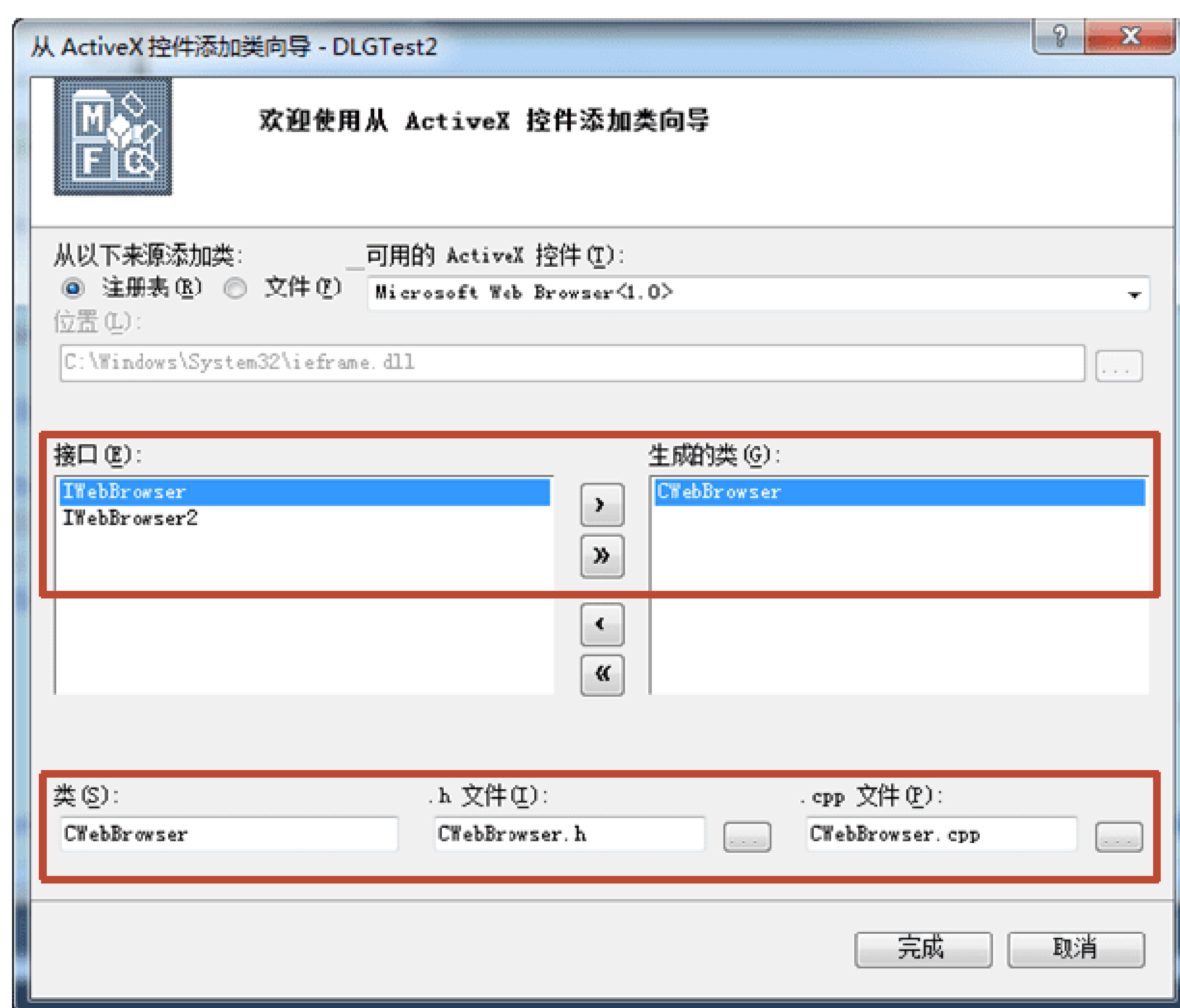


图 7-21　选择加入的包装类

从对话框中插入 ActiveX 控件的操作步骤如下：

（1）在对话框编辑器中右击，在弹出的快捷菜单中选择“插入 ActiveX 控件”，弹出如图 7-22 所示的界面。

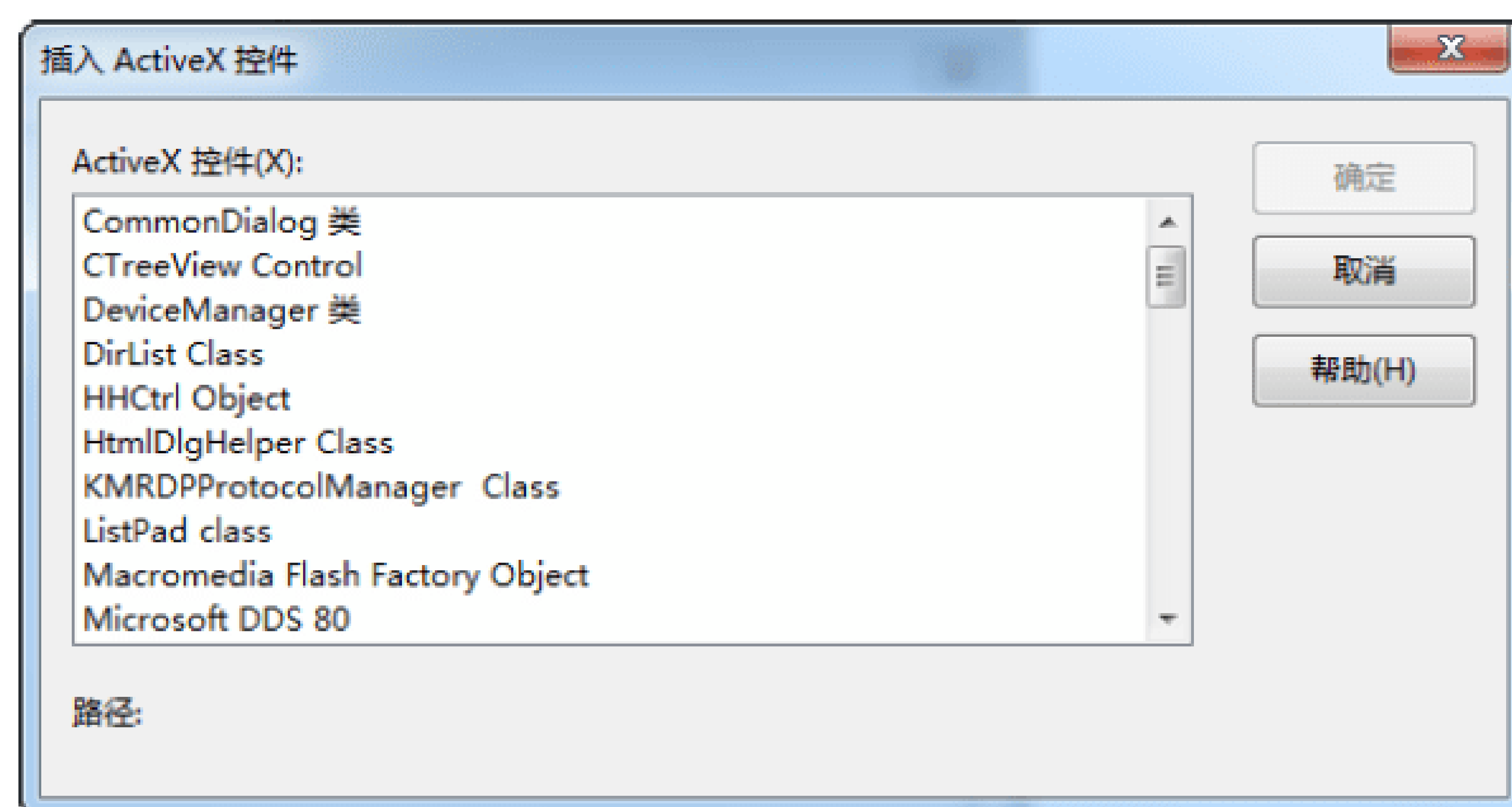


图 7-22　在对话框中插入 ActiveX 控件

（2）在左边的列表框中选择要插入的 ActiveX 控件，单击“确定”按钮。

（3）类向导会生成包装类。

7.8.6　MFC 程序中 ActiveX 控件的使用

在 MFC 程序中，通过在 InitInstance()函数中添加以下代码可以添加对 ActiveX 控件的使用支持。

```
AfxEnableControlContainer();                //启用对 ActiveX 控件的支持
```

按照 7.8.6 小节中介绍的方法，向工程中添加要用的 ActiveX 控件后，像使用 Windows 标准控件的过程一样使用就可以。在后面介绍数据库篇时，会涉及到 ActiveX 控件的使用。

7.9　本 章 小 结

Windows 标准控件在 Windows 操作系统中有广泛的应用，是界面程序中不可缺少的部分。本章主要介绍了其中的几种，包括按钮控件、ActiveX 控件等。因为每种控件都有其自身的特殊属性和方法，因此，本章只起到抛砖引玉的作用，在此基础上，读者应该可以触类旁通、举一反三。第 8 章将介绍 MFC 的一些常用类。

7.10　习　　题

1．创建基于对话框的应用程序 Dlg，添加 3 个控件：编辑框、按钮和静态控件。设置界面如图 7-23 所示。当单击 Button1 按钮时，会将编辑框中的文本显示在静态控件上。程序的运行效果如图 7-24 所示。



图 7-23　对话框界面设计

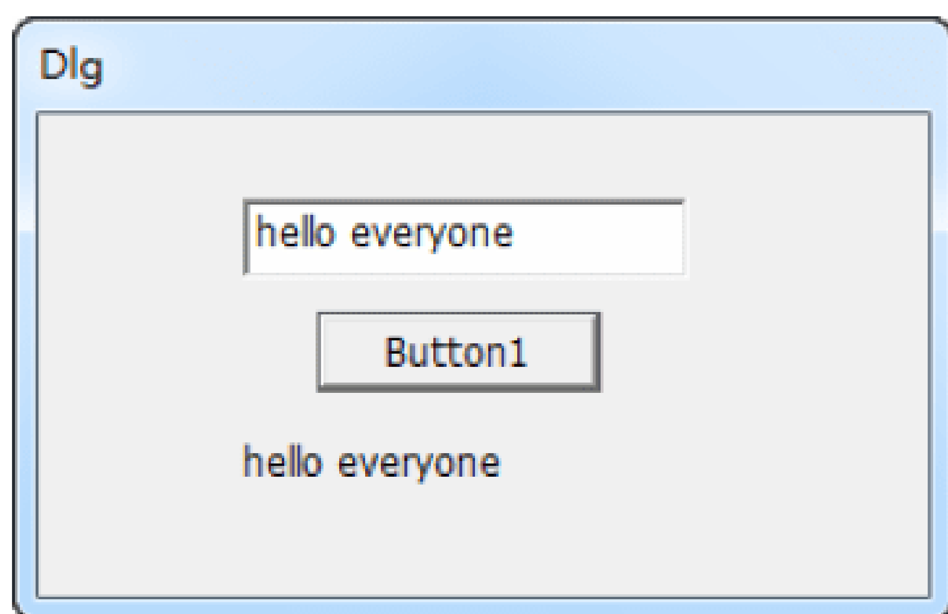


图 7-24　程序的运行效果

【思路】编辑框类 CEdit 和静态控件类 CStatic 都继承自类 CWnd，所以它们也继承了 CWnd 的成员函数 GetWindowText()和函数 SetWindowText()，前者用来获取控件文本的内容，后者用来设置控件显示的文本内容。

2．创建基于对话框的应用程序 Dlg，添加两个控件：列表框和组合框。在对话框初始化的时候为这两个控件填充内容（星期），运行效果如图 7-25 所示。

【思路】可以参考 7.5.4 小节和 7.5.8 小节的实例。为列表框和组合框添加字符串时用到的函数是 AddString()。

3．创建基于对话框的应用程序 Dlg，添加两个控件：列表视图和树形视图。在对话框初始化的时候为这两个控件填充内容（数字），运行效果如图 7-26 所示。

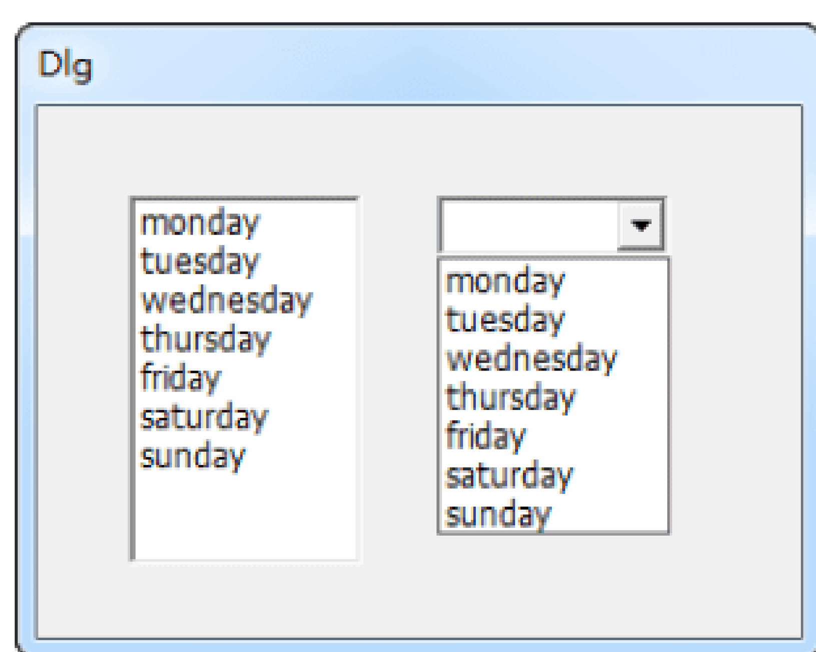


图 7-25　习题 2 程序运行效果

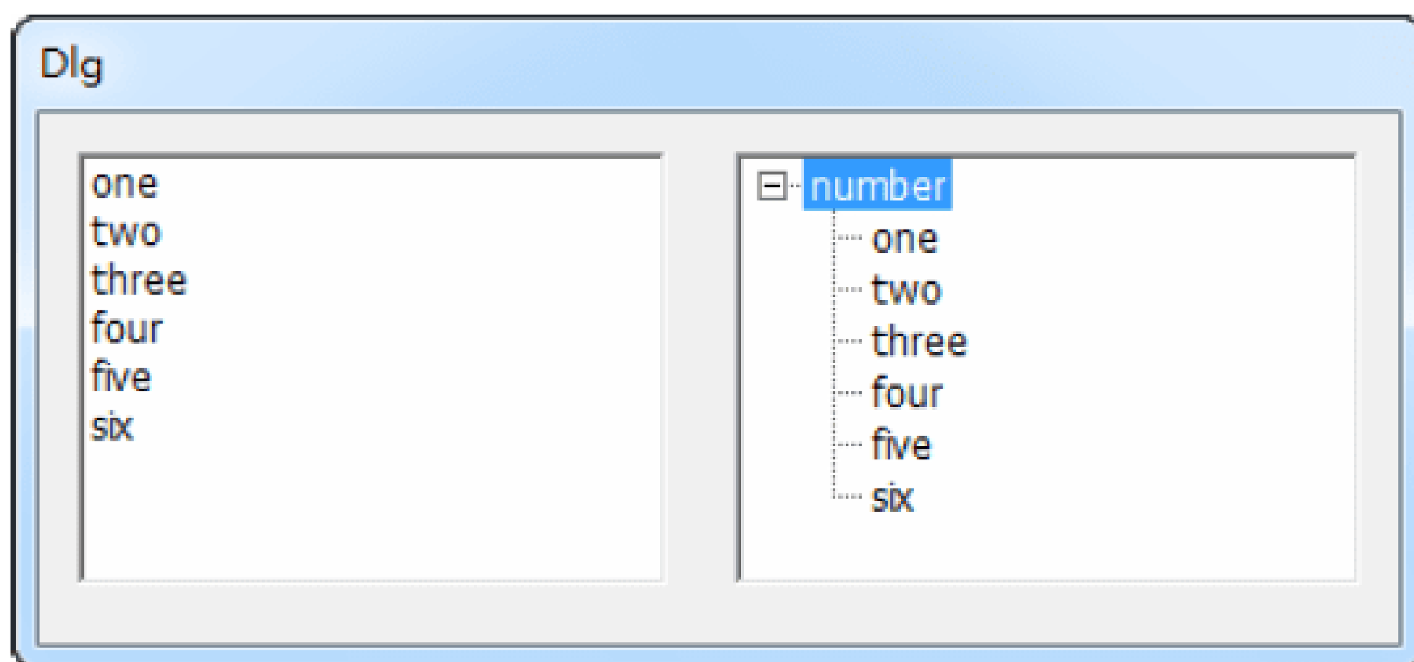


图 7-26　习题 3 程序运行效果

【思路】类 ClistCtrl 和 CTreeCtrl 都有一个成员函数 InsertItem()，可以用来为视图控件填充字符串。

第 8 章　MFC 的一些常用类

MFC 为应用程序的开发封装了一些常用的类。这些类在各种应用程序中都会遇到。CString 类是 MFC 中用于处理字符串的类，封装了常用的字符串操作。MFC 通过数组和链表类实现 MFC 的集合类。对于特殊类型——日期时间型，MFC 通过 CTime 类实现。CFile 类封装有关文件的读写等操作。MFC 还提供了异常类来提高程序的健壮性。本章就分别介绍这几种常用类。

8.1　字符串类（CString）

CString 类是用于存储和管理字符数组的类。CString 类在内存中完成字符串的连接和比较等操作。由于它对字符串操作时自动处理存储空间的大小，不需要开发人员手动处理内存的分配等问题，因此，大大简化了开发人员维护字符串的工作量。本节将介绍 CString 类的使用方法。

8.1.1　创建 CString 对象

要使用字符串类 CString，首先要创建类对象。MFC 为 CString 类提供了多种创建方式，每种创建方式都提供对应的构造函数，使用指定的数据初始化新建的 CString 对象。因为构造函数会将输入的数据复制到新分配的存储空间中，所以，在初始化时有可能会发生异常，因此，程序应该做好异常处理。

CString 类提供了如下几种形式的构造函数：

```
CString( );                                    //不带参数的构造函数
CString( const CString& stringSrc );           //使用 CString 参数的构造函数
CString( TCHAR ch, int nRepeat = 1 );          //使用重复字符的构造函数
CString( LPCTSTR lpch, int nLength );          //带指定长度字符串的构造函数
CString( const unsigned char* psz );           //带以 NULL 结束的字符串的构造函数
CString( LPCWSTR lpsz );                   //带以 NULL 结束的多字符集字符串的构造函数
CString( LPCSTR lpsz );                    //带以 NULL 结束的单字符集字符串的构造函数
```

从上面也可以看出，CString 类的构造函数可以作为转换函数，实现与 const char*和 LPCTSTR 等数据类型之间的转换。

8.1.2　CString 类的成员函数

除了前面介绍的用于创建 CString 对象的构造函数外，CString 类还包括多个成员函数

实现字符串操作，包括赋值、连接、比较、转换、数据存取、序列化和格式化等，如表 8-1 所示。

表 8-1　CString类的成员函数

函　数　名	说　　明
=操作符	为 CString 对象赋值
+操作符	连接两个字符串，并返回连接后的新字符串
+=操作符	连接两个字符串，并将连接后的新字符串赋值给操作符左边的变量
==、<、>、<>等比较操作符	比较操作符，区分大小写
Compare()	比较两个字符串，区分大小写
CompareNoCase()	比较两个字符串，不区分大小写
Collate()	使用本地化设置比较两个字符串，区分大小写
CollateNoCase()	使用本地化设置比较两个字符串，不区分大小写
Find()	从字符串头开始查找字符或子字符串
ReverseFind()	从字符串尾开始查找字符或子字符串
FindOneOf()	从字符串头开始查找第一个匹配的字符或子字符串
MakeUpper()	将字符串中的多有字符转换成大写
MakeLower()	将字符串中的多有字符转换成小写
MakeReverse()	反转字符串中的字符
Replace()	替换字符串中指定的字符
Remove()	移除字符串中指定的字符
Insert()	在字符串中指定的位置上插入一个字符或另一个子字符串
Delete()	从字符串中删除一个字符或一个字符串
Format()	使用 sprintf 方式格式化字符串
FormatV()	使用 vsprintf 方式格式化字符串
TrimLeft()	取消字符串前面的空格
TrimRight()	取消字符串后面的空格
FormatMessage()	格式化消息字符串
Mid()	截取字符串中间的数据
Left()	截取字符串左边的数据
Right()	截取字符串右边的数据
SpanIncluding()	截取字符串中包含指定字符的部分
SpanExcluding()	截取字符串中不包含指定字符的部分
operator <<	插入字符串对象
operator >>	提取字符串对象
GetLength()	返回 CString 对象的字符串数。对于多字符集，也是按照 8 位字符计数，也就是每个多字节字符算作两个字符
IsEmpty()	判断 CString 对象是否不包含任何字符
Empty()	清空字符串内容
GetAt()	返回指定位置的字符
[]	返回指定位置的字符，作用与 GetAt()函数相同
SetAt()	设置指定位置的字符
LPCTSTR 操作符	直接访问存储在 CString 对象中的字符串
GetBuffer()	返回字符串对象的字符指针

续表

函　数　名	说　　明
GetBufferSetLength()	返回字符串中指定长度的字符串的指针
ReleaseBuffer()	释放由 GetBuffer()函数获取的对缓冲区的控制
FreeExtra()	移除字符串对象以前分配给字符串前面的字符
LockBuffer()	锁定字符缓冲区，并关闭引用计数
UnlockBuffer()	释放字符缓冲区，并打开引用计数
AllocSysString()	从字符串对象复制数据到新创建的 BSTR 对象的变量中
SetSysString()	复制字符串对象的数据到一个存在的 BSTR 对象中
LoadString()	从 Windows 资源中装载一个存在的字符串对象
AnsiToOem()	将字符串对象中的字符从 ANSI 字符集转换成 OEM 字符集
OemToAnsi()	将字符串对象中的字符从 OEM 字符集转换成 ANSI 字符集

8.1.3　CString 类的常用操作

8.1.2 小节列出了 CString 的成员函数，本小节就结合这些成员函数具体讲述 CString 类的常用操作。CString 类通过提取操作符和插入操作符实现对序列化的支持，这两个函数的原型为：

```
//提取操作符
friend CArchive& operator <<( CArchive& ar, const CString& string );
//插入操作符
friend CArchive& operator >>( CArchive& ar, CString& string );
//提取操作符
friend CDumpContext& operator <<( CDumpContext& dc, const CString& string );
```

提取操作符<<将 CString 对象序列化到文件等对象中，插入操作符>>从文档对象中反序列化 CString 对象。

CString 类提供 GetLength()函数，返回字符串的长度。CString 类以 TCHAR 为基本数据类型存储字符串。也就是说，在 Unicode 字符编码方式下，CString 类使用 16 位字符存储字符串；否则，CString 存储的是 8 位的字符。默认情况下，CString 类也是支持双字节字符集（double-byte character sets，DBCS）的，此时使用此方法返回的字符串长度是以 8 位字符为基准的，即每个字节算作一个字符。其函数原型为：

```
int GetLength( ) const;          //获取字符串长度，返回值为字符串的长度
```

CString 类提供了 Empty()函数和 IsEmpty()函数，分别用于清空字符串和判断字符串对象是否没有数据。其函数原型为：

```
//清空字符串
void Empty( );
//判断字符串是否为空。如果对象长度为 0，返回 true，否则返回 false
BOOL IsEmpty( ) const;
```

CString 类提供了一组进行字符串比较的函数，主要有 4 个函数，其函数原型为：

```
int Collate( LPCTSTR lpsz ) const;          //比较字符串
int CollateNoCase( LPCTSTR lpsz ) const;    //不区分大小写的比较字符串
```

```
int Compare( LPCTSTR lpsz ) const;          //使用单字符集比较字符串
//使用单字符集不区分大小写的比较字符串
int CompareNoCase( LPCTSTR lpsz ) const;
```

其中，参数 lpsz 为要比较的字符串。如果两个字符串相等，则返回 0；如果字符串对象小于 lpsz 参数指定的字符串，则返回值<0；如果字符串对象大于 lpsz 参数指定的字符串，则返回值>0。其中，Collate()函数和 Compare()函数在进行字符串比较时是区分大小写的，而 CollateNoCase()函数和 CompareNoCase()函数在进行字符串比较时是不区分大小写的。Collate()函数和 Compare()函数的区别在于，Collate()函数是基于字符集设置进行比较，支持多字符集的比较，而 Compare()函数是基于单字符集进行比较的。

CString 类提供了一组进行字符串操作的函数，包括增加字符、删除字符和插入字符，其函数原型为：

```
int Delete(                 //返回值为成功删除的字符的个数
     int nIndex,            //开始删除字符的第一个字符的索引
     int nCount = 1 );      //要删除的字符个数
int Insert(                 //在指定位置插入字符，返回值为增加字符后的字符串的长度
     int nIndex,            //要在其前增加字符的索引
     TCHAR ch );            //要增加的字符
int Insert(                 //在指定位置插入字符串，返回值为增加字符后的字符串的长度
     int nIndex,            //要在其前增加字符的索引
     LPCTSTR pstr );        //要增加的字符串的指针
```

CString 类提供了一组在字符串中查找的函数。Find()函数表示在字符串对象中正向查找字符或字符串，ReverseFind()函数表示在字符串对象中反向查找字符，即从字符串尾开始查找。其函数原型为：

```
//查找指定字符 ch，返回索引位置值
int Find( TCHAR ch ) const;
//查找指定字符串 lpszSub，返回开始的索引位置
int Find( LPCTSTR lpszSub ) const;
//从 nStart 位置开始查找 ch 字符
int Find( TCHAR ch, int nStart ) const;
//从 nStart 位置开始查找 pstr 字符串
int Find( LPCTSTR pstr, int nStart ) const;
//反向查找字符串中的 ch 字符
int ReverseFind( TCHAR ch ) const;
//返回字符串中是否包含 lpszCharSet 中的任何一个字符以及这个字符的位置
int FindOneOf(
    LPCTSTR lpszCharSet     //要查找的字符的字符串，其中的字符无顺序而言
     ) const;
```

这组函数的返回值表示查找到的字符或字符串的第一个字符在查找字符串中的索引，–1 表示字符没有查找到。

CString 提供了一组截取字符串数据的函数，其函数原型为：

```
//获取指定索引处的字符，返回值为 TCHAR
TCHAR GetAt(
    int nIndex
) const;
//获取字符串中 nIndex 索引处的字符，大于 0 并小于 GetLength 返回值
TCHAR operator []( int nIndex ) const;
//将字符串作为数组操作，但是此操作是只读操作
```

CString 类提供了一组操作缓冲区数据的函数。FreeExtra()函数释放字符串对象以前分配的但是现在不再使用的内存，按照字符串的长度为字符串对象重新分配缓冲区，这样可以减少内存的无效使用。GetBuffer()函数和 ReleaseBuffer()函数分别用于获取字符串缓冲区和释放字符串缓冲区。GetBufferSetLength()函数则用于设置字符串缓冲区长度。LockBuffer()函数和 UnlockBuffer()函数分别用于锁定字符串缓冲区和解锁字符串缓冲区。函数原型如下：

```
void FreeExtra( );        //释放字符串占用的内存
LPTSTR GetBuffer(         //获取字符串缓冲区，返回字符串对象的内部字符缓冲区指针
    int nMinBufLength     //表示字符缓冲区中最小的字符数
);
//设置字符串长度，返回字符串对象的内部字符缓冲区指针
LPTSTR GetBufferSetLength(
    int nNewLength        //表示设置的缓冲区字符的精确长度，-1 表示当前字符串长度
);
//释放字符串缓冲区
void ReleaseBuffer( int nNewLength = -1 );
//锁定字符串，保护数据不被其他字符串引用，也不能引用其他字符串
LPTSTR LockBuffer( );
void UnlockBuffer( );   //解锁字符串
```

为了节约存储空间，CString 类允许两个字符串共享相同的值或者存储区空间。因此要直接操作存储区中的数据时，应该在操作前使用 LockBuffer()函数锁定存储区，并在操作后使用 UnLockBuffer()函数解锁存储区。

CString 类提供了一组数据截取函数，返回值为获取的字符串副本。函数原型如下：

```
CString Left( int nCount ) const;  //获取字符串左边 nCount 个字符
CString Right( int nCount ) const; //获取字符串右边 nCount 个字符
CString Mid( int nFirst ) const;   //获取从 nFirst 位置开始的字符串
CString Mid( int nFirst, int nCount ) const;
                                   //获取从 nFirst 位置开始的 nCount 个字符
CString SpanExcluding( LPCTSTR lpszCharSet ) const;
                                   //返回排除 lpszCharSet 字符集的字符串
CString SpanIncluding( LPCTSTR lpszCharSet ) const;
                                   //返回包含 lpszCharSet 字符集的字符串
```

8.1.4　CString 的格式化与类型转换

CString 类提供了一组格式化字符串的函数，其函数原型为：

```
void Format( LPCTSTR lpszFormat, ... );     //按照格式化字符串格式化
void Format( UINT nFormatID, ... );         //按照格式化 ID 格式化字符串
void FormatV( LPCTSTR lpszFormat, va_list argList );
                                            //使用参数格式化字符串
//格式化消息字符串
void FormatMessage( LPCTSTR lpszFormat, ... );
//按照格式化 ID 格式化消息字符串
void FormatMessage( UINT nFormatID, ... );
```

其中，lpszFormat 参数是一个格式化控制字符串，nFormatID 参数是一个包含格式化控

制字符串资源的标识，argList 参数是一个传入的参数列表。此函数的功能与 sprintf()函数相同，会将第一个参数后的所有可选参数按照提供的格式化控制字符串的形式写入字符串对象中。此函数对于传入的参数个数不固定的情况非常有用。函数没有返回值。

注意：不能在此函数中使用字符串对象本身作为可选参数写入字符串对象，因为这样会造成循环递归，发生不可预知的错误。

CString 类提供了一组格式化字符串头和字符串尾的函数。函数原型为：

```
void TrimLeft( );                //去除左边的空格，包括换行、空格和 tab 字符
void CString::TrimLeft( TCHAR chTarget );
                                 //去除字符串左边的 chTarget 字符
void CString::TrimLeft( LPCTSTR lpszTargets );
                                 //去除字符串左边的 lpszTargets 字符串
void TrimRight( );               //去除右边的空格，包括换行、空格和 tab 字符
void CString::TrimRight( TCHAR chTarget );
                                 //去除字符串右边的 chTarget 字符
void CString::TrimRight( LPCTSTR lpszTargets );
                                 //去除字符串右边的 lpszTargets 字符串
```

CString 类提供了一组字符大小写转换函数，完成字符串的大小写转换。函数原型为：

```
void MakeLower( );               //将字符串中的字符全部转换成小写字母
void MakeUpper( );               //将字符串中的字符全部转换成大写字母
```

CString 类提供了一组数据处理函数。函数原型为：

```
void MakeReverse( );     //反转字符串中的字符，即按照相反的顺序重新排列字符顺序
int CString::Remove( TCHAR ch );
                         //移除字符串中的所有 ch 字符，返回移除的字符数目
//替换 chOld 字符为 chNew 字符
int Replace( TCHAR chOld, TCHAR chNew );
int Replace( LPCTSTR lpszOld, LPCTSTR lpszNew );
                         //替换 lpszOld 字符串为 lpszNew 字符串
//设置 nIndex 索引处的字符为 ch 字符
void SetAt( int nIndex, TCHAR ch );
```

其中，Replace()函数可以使用指定字符或字符串替换原有字符串中指定的字符或字符串，如果被替换和替换的字符串的长度不相等，则字符串的长度有可能发生变化，同时替换是区别大小写的。

CString 类实现类型转换有 3 种方法。第一种方法是通过构造函数实现，这在 8.1.1 小节中介绍过。第二种方法是通过赋值操作符，以下代码是 CString 类的赋值操作符原型。

```
const CString& operator =( const CString& stringSrc );
                                       //使用 CString 参数的赋值操作符
const CString& operator =( TCHAR ch ); //使用字符参数的赋值操作符
const CString& operator =( const unsigned char* psz );
                                       //使用字符串参数的赋值操作符
const CString& operator =( LPCWSTR lpsz );
                                       //使用多字符集字符串参数的赋值操作符
const CString& operator =( LPCSTR lpsz );
                                       //使用单字符集字符串参数的赋值操作符
const CString& operator +=( const CString& string );
                                       //使用字符串参数的加法赋值操作符
```

```
const CString& operator +=( TCHAR ch );//使用字符参数的加法赋值操作符
const CString& operator +=( LPCTSTR lpsz );
                                        //使用字符串参数的加法赋值操作符
```

CString 类的赋值操作符实现了对已定义 CString 对象的重新赋值功能，作用与构造函数相似，如果目的字符串有足够存储空间存储新分配的值，则系统不会为其分配新内存，如果存储空间不够，则系统会为其重新分配存储空间。其中+=操作符则提供了连接赋值的功能，即将数据连接后将其赋值给 CString 对象。

第三种类型转换的方法是使用 CString 类的成员函数。其函数原型为：

```
BOOL LoadString( UINT nID );    //装载字符串，nID 参数为 Windows 字符串资源的 ID
//从 OEM 类型装换成 Ansi 类型，如果定义了_UNICODE 宏，则此函数无效
void OemToAnsi( );
//将字符串 pbstr 复制到 BSTR 类型中，并返回
BSTR SetSysString( BSTR* pbstr ) const;
operator LPCTSTR ( ) const;     //将 CString 对象转换成 LPCTSTR 对象
```

需要注意的是，LPCTSTR 操作符返回的对象指针有可能发生变化，所以在使用此函数获取字符串指针后，在下次使用它对字符串进行操作时，指针可能已经不指向正确的字符串了。因此，尽量不要使用 LPCTSTR 操作符返回的对象对字符串进行写操作。

8.1.5　CString 使用实例

以下代码是 CString 类的使用实例，其中分别演示了 Mid()、Left()、Right()、GetLength()和 Insert()等函数的使用。程序源代码如下：

```
01  #include <iostream>
02  #include <atlstr.h>
03  using namespace std;
04
05  int main()
06  {
07      //定义字符串
08      CString s( _T("abcdef") );
09      cout << "初始字符串=" << (LPCTSTR)s << endl;
10      cout << "第三个字符开始的三个字符是否为'cde'="
11          << (s.Mid( 2, 3 ) == _T("cde")) << endl;
12      cout << "左边两个字符是否为'ab'="
13          << (s.Left(2) == _T("ab") ) << endl;
14      cout << "右边两个字符是否为'ef'="
15          << (s.Right(2) == _T("ef") ) << endl;
16      //定义字符串，并初始化
17      CString str("Welcome to Beijing");
18      cout << "初始字符串=" << (LPCTSTR)str
19          << ";字符串长度=" << str.GetLength() << endl;
20      //在字符串的第 7 个字符后添加字符串
21      int n = str.Insert(7, " is");
22      cout << "添加'is '后的字符串内容=" << (LPCTSTR)str
23          << ";字符串长度=" << str.GetLength() << endl;
24      return 0;
25  }
```

上面程序定义了字符串变量 s，输出 s 的值。第 11 行语句判断第 3 个字符开始的 3 个字符是否为 cde，第 13 行语句判断最左边两个字符是否为 ab，第 15 行语句判断最右边两个字符是否为 ef。接下来定义字符串变量 str 并输出其值和长度，并调用 Insert 语句插入字符，最后输出插入字符后字符串的内容及长度。程序运行的效果如图 8-1 所示。

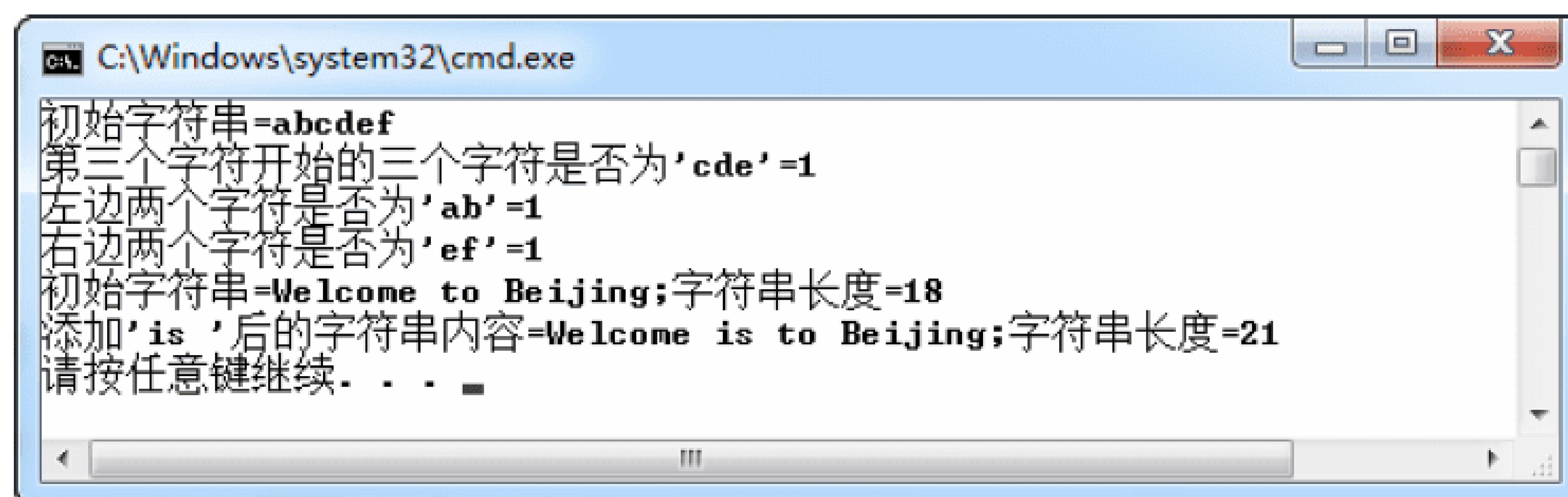


图 8-1　CString 示例运行效果

8.2　集　合　类

集合是存储多个数据结构相同的对象的容器。MFC 提供了顺序结构的数组类和指针结构的链表类实现集合的功能。并且在这两种结构的基础上，扩展了常用数据类型的数组和数组模板类，以及常用数据类型的链表和链表模板类。本节将介绍 MFC 的集合类的使用。

8.2.1　数组类

数组是 C++语言中基本的数据类型，MFC 封装了 CArray 类实现数组的功能。并且实现了动态缩减和增加数组长度的功能，索引也是从 0 开始。类声明如下：

```
template< class TYPE, class ARG TYPE >
class CArray : public CObject
```

其中，模板参数 TYPE 指定存储在数组中的对象类型，也是 CArray 类中返回的参数的类型。模板参数 ARG_TYPE 指定用于访问存储在数组中的对象的参数类型，通常是 TYPE 的引用，也是传入 CArray 的参数。CArray 类既可以固定数组大小，也可以动态添加元素扩展数组大小。不管数组元素中的值是什么，Carray 都会连续分配内存。与 C 中的数组一样，使用索引访问 CArray 元素时，索引值必须为常数，且必须在数组大小范围内。除了封装了数组的功能，CArray 类还提供了一组数组操作成员函数，如表 8-2 所示。

表 8-2　CArray类的数组操作成员函数

函　数　名	功　　能
GetSize()	获取数组中的元素个数
GetUpperBound()	返回数组最大的有效索引
SetSize()	设置数组中包含的元素数目
FreeExtra()	释放当前数组范围下，不使用的内存
RemoveAll()	从数组中移除所有的元素
GetAt()	返回指定索引处的元素值

续表

函　数　名	功　　能
SetAt()	设置指定索引处的值，不能使用此函数增加元素，只能修改已经存在的元素值
ElementAt()	返回指向数组中指定索引处的元素的引用
GetData()	允许访问数组中的元素，可以为 NULL
SetAtGrow()	设置指定索引处的值，与 SetAt()函数不同。如果需要，此函数可以自动增加数组大小
Add()	向数组尾添加元素，如果需要，会自动增加数组大小
Append()	添加其他数组中的元素到数组中，如果需要，会自动增加数组大小
Copy()	复制其他数组中的元素到数组中，如果需要，会自动增加数组大小
InsertAt()	在指定索引处插入元素或其他数组中的所有元素
RemoveAt()	移除指定索引处的元素
operator []()	设置或获取指定索引处的元素

技巧： 在使用数组前，最好使用 SetSize()确定数组大小，系统会为其分配空间。如果不调用此函数，增加元素时，有时会重新分配空间，并频繁地进行复制，这样会降低程序效率，并产生很多内存碎片。

CArray 是个模板类，可以定义数组中存放的类型，可以存放 C++中定义的类型的对象，也可以存放用户自定义类型的对象。为了简化操作，MFC 提供了存放常用类型的对象的数组类。这些类的成员方法与 CArray 模板类的成员函数类似，只是指定了具体的元素类型。MFC 中从 CArray 类派生的数组类有以下几个。

- CByteArray：存储 BYTE 类型元素的数组。
- CDWordArray：存储 DWORD 类型元素的数组。
- CObArray：存储 CObject 类对象或派生自 CObject 类的对象的数组。
- CPtrArray：存储指向 void 的指针元素的数组。
- CUIntArray：存储 UINT 类型元素的数组。
- CWordArray：存储 WORD 类型元素的数组。
- CStringArray：存储 CString 对象元素的数组。

8.2.2　数组类的使用实例

8.2.1 小节介绍了模板类 CArray 及其派生类，本小节以一个实例介绍如何使用数组类 CArray 存储自定义对象。代码如下：

```
01  CArray<CPoint,CPoint> m_ptArray;//定义点元素数组对象
02  void PrintArray()
03  {
04      for (int i = 0 ;i < m_ptArray.GetSize(); i++)
05      {
06          cout << "第" << i+1 << "个元素=(" << m_ptArray[i].x
07                << "," << m_ptArray.GetAt(i).y << ")" << endl;
08      }
09  }
10  int _tmain()
```

```
11  {
12      ...
13          CPoint ptA(0, 0), ptB(20, 50), ptC(120,300);//构造 CPoint 对象
14          m_ptArray.Add(ptA);                          //增加元素
15          m_ptArray.Add(ptB);
16          m_ptArray.Add(ptC);
17
18          cout << "初始 CArray 对象中共有" << m_ptArray.GetSize()
19              << "个元素：" << endl;
20          PrintArray();
21
22          CPoint ptD(800,600);
23          m_ptArray[1] = ptD;                 //修改元素值
24          cout << endl << "修改第二个元素后 CArray 对象中共有"
25              << m_ptArray.GetSize() << "个元素：" << endl;
26          PrintArray();
27
28          m_ptArray.RemoveAt(0);              //移除第一个元素值
29          cout << endl << "删除第一个元素后 CArray 对象中共有"
30              << m_ptArray.GetSize() << "个元素：" << endl;
31          PrintArray();
32
33          m_ptArray.RemoveAll();              //移除所有元素值
34          cout << endl << "删除所有元素后 CArray 对象中共有"
35              << m_ptArray.GetSize() << "个元素：" << endl;
36          PrintArray();
37      ...
38  }
```

上面的代码首先定义了 CArray 类型数组对象 m_ptArray，用于存储 CPoint 类型的对象。接着定义了 PrintArray()函数用于输出数组对象中的所有元素。在_tmain 主函数中，首先向 m_ptArray 对象中添加 3 个点元素，并输出初始情况下数组对象中的元素值。接着使用新的点元素值修改数组中的第二个元素，并输出修改元素值后数组对象中的元素值。然后使用 RemoveAt()函数删除数组对象中的第一个元素，并将操作后的数组对象中的元素值打印出来。最后，调用 RemoveAll()函数删除数组对象中的所有元素，并输出数组对象的取值情况。程序运行效果如图 8-2 所示。

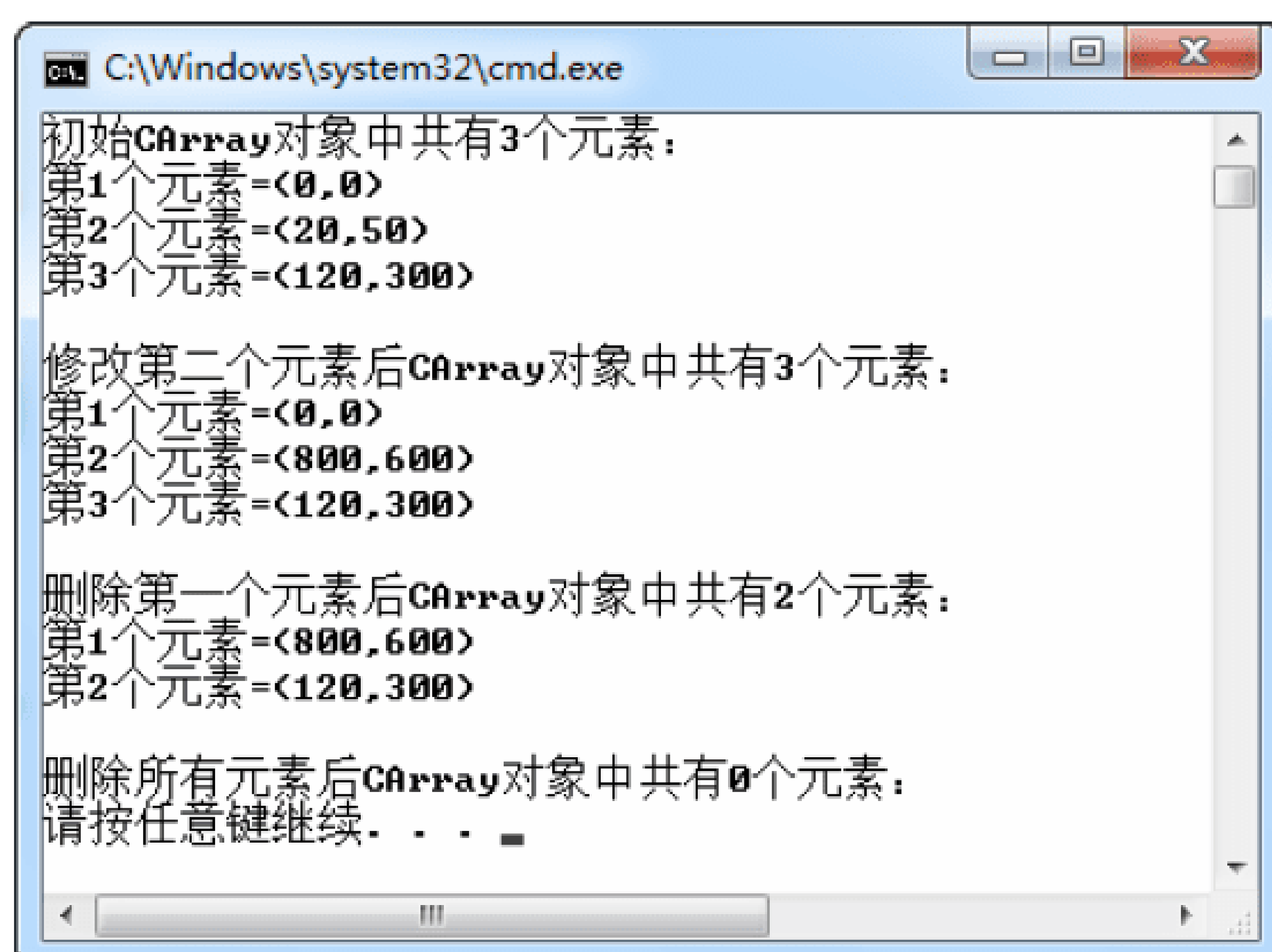


图 8-2　数组类使用实例运行效果

8.2.3　链表类

CList 类与 CArray 类的实现方式类似，区别在于，CArray 实现顺序存储的数组，而 CList 类实现对象的链表存储方式，其是双向指针链表。其中使用 POSITION 类型的变量表示链表中元素的位置，但是 POSITION 与索引的类型是不同的，可以看作书签。链表类的声明如下：

```
template< class TYPE, class ARG TYPE >
class CList : public CObject
```

其中，模板参数 TYPE 指定存储在链表中的对象类型，也是 CList 类中返回的参数的类型。模板参数 ARG_TYPE 指定用于访问存储在链表中的对象的参数类型，通常是 TYPE 的引用，也是传入 CList 的参数。CList 类相对于 CArray 类的优点与链表存储相对于顺序存储的优点是相同的，在链表中在链表头、链表尾和已知 POSITION 位置插入元素和删除元素的速度都非常快，即 CList 类对于频繁插入删除元素的情况非常适用。CList 提供了一组操作链表和链表元素的成员函数，以简化链表结构的维护工作量，如表 8-3 所示。

表 8-3　链表类CList成员函数

函　数　名	函 数 功 能
GetHead()	获取链表的头元素，即第一个元素
GetTail()	获取链表的尾元素，即最后一个元素
RemoveHead()	移除链表的头元素
RemoveTail()	移除链表的尾元素
AddHead()	在链表头处增加新元素或将其他链表中的元素增加到链表头
AddTail()	在链表尾处增加新元素或将其他链表中的元素追加到链表尾
RemoveAll()	移除链表中的所有元素
GetHeadPosition()	返回链表头的位置值
GetTailPosition()	返回链表尾的位置值
GetNext()	相对于当前元素，获取链表中的下一个元素
GetPrev()	相对于当前元素，获取链表中的前一个元素
GetAt()	获取指定位置值的元素
SetAt()	修改指定位置值的元素值
RemoveAt()	移除指定位置值的元素值
InsertBefore()	在指定位置值前插入一个新元素
InsertAfter()	在指定位置值后插入一个新元素
Find()	查找指定元素所在的位置
FindIndex()	返回基于 0 索引的元素的位置值
GetCount()	获取链表中元素的数目
IsEmpty()	测试链表是否为空

从上面的成员函数可以看出，CList 类与 CArray 类不同的是，它不能简单地通过索引存取数据。CList 对象必须将索引值转换成位置标签值，通过位置标签值存取其中的数据。这是因为链表类实现的是链表结构，它的存储区域不是连续的，而是通过指针链接起来的。因此，必须判断出具体位置才可以存取。

CList 是个模板类，可以定义链表中存放的类型，可以存放 C++中定义的类型的对象，也可以存放用户自定义类型的对象。为了简化操作，MFC 提供了存放常用类型的对象的链

表类。这些类的成员方法与 CList 模板类的成员函数类似，只是指定了具体的元素类型。MFC 中从 CList 类派生的链表类有以下 3 个。

- CObList 类：表示用于存放 CObject 对象的链表。
- CPtrList 类：表示用于存放指针对象的链表。
- CStringList 类：表示用于存放 CString 对象的链表。

8.2.4　链表类的使用实例

8.2.3 小节介绍了模板类 CList 及其派生类，本小节以一个实例介绍如何使用链表类 CList 存储整型数据以及 CStringList 的使用。代码如下：

```
01  CStringList m_stringList;                    //定义字符串型链表
02  void PrintList()
03  {
04      cout << "当前字符串链表中共有" << m_stringList.GetCount()
05            << "个元素" << endl;
06      if (m_stringList.IsEmpty())
07          return;
08      POSITION pos = m_stringList.GetHeadPosition();
09      int i = 0;
10      while (pos != NULL)
11      {
12          cout << "第" << i+1 << "个元素='"
13                << (LPCTSTR)m_stringList.GetAt(pos) << "'" << endl;
14          m_stringList.GetNext(pos);
15      }
16      cout << endl;
17  }
18
19  int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
20  {
21      ...
22          CList<int,int> m_intList;                //定义整型链表
23          m_intList.AddHead(2008);
24          m_intList.AddHead(2009);
25          cout << "整数链表中共有" << m_intList.GetCount()
26                << "个元素" << endl;
27          POSITION posInt = m_intList.GetHeadPosition();
28          int i = 0;
29          while (posInt != NULL)
30          {
31              cout << "第" << i++ << "个整数元素="
32                    << m_intList.GetAt(posInt) << endl;
33              m_intList.GetNext(posInt);
34          }
35
36          for (i = 0;i < 3; i++)
37          {
38              CString element;
39              element.Format("这是第%d个元素", (i+1));
40              m_stringList.AddTail(element);   //向链表尾添加元素
41          }
42          PrintList();
43
44          POSITION pos = m_stringList.FindIndex(1);
```

```
45          if (pos != NULL)
46          {
47              cout << "执行操作----使用 InsertBefore 函数在当前链表的";
48              cout << "第二个元素前插入新元素" << endl;
49              m_stringList.InsertBefore(pos, "在第二个元素前插入的新元素");
50              PrintList();
51
52              cout << "执行操作----使用 SetAt 函数修改第二个元素值" << endl;
53              m_stringList.SetAt(pos, "修改了第二个元素");
54              PrintList();
55
56              pos = m_stringList.FindIndex(1);
57              cout << "执行操作----使用 RemoveAt 函数移除第二个元素" << endl;
58              m_stringList.RemoveAt(pos);       //移除最后一个元素
59              PrintList();
60
61              cout << "执行操作----使用 RemoveAll 函数移除所有元素" << endl;
62              m_stringList.RemoveAll();         //移除链表中的所有元素
63              PrintList();
64      ...
65  }
```

上面的代码定义了字符串链表对象 m_stringList 和输出字符串链表中的元素值和元素个数的 PrintList()函数。在_tmain()函数中演示了链表的常用操作，包括增加元素、删除元素、修改元素和移除所有元素等操作。在初始化了 m_intList 对象后，为其增加元素，并输出元素内容。为 m_stringList 对象初始化数据后，输出初始化的数据。然后，使用 FindIndex()函数获取第二个元素对应的位置值，并调用 InsertBefore()函数在其前面插入新元素，接着使用 SetAt()函数修改第二个元素值，注意这里的第二个元素还是指向原来的第二个元素，即现在的第三个元素，因为链表是以位置值为关键值，而不像数组一样使用索引作为关键值。因此，要移除当前的第二个元素之前，需要重新调用 FindIndex()函数获取当前第二个元素的位置值，并调用 RemoveAt()函数移除此元素，最后调用 RemoveAll()函数移除链表中的所有元素。程序运行的效果如图 8-3 所示。

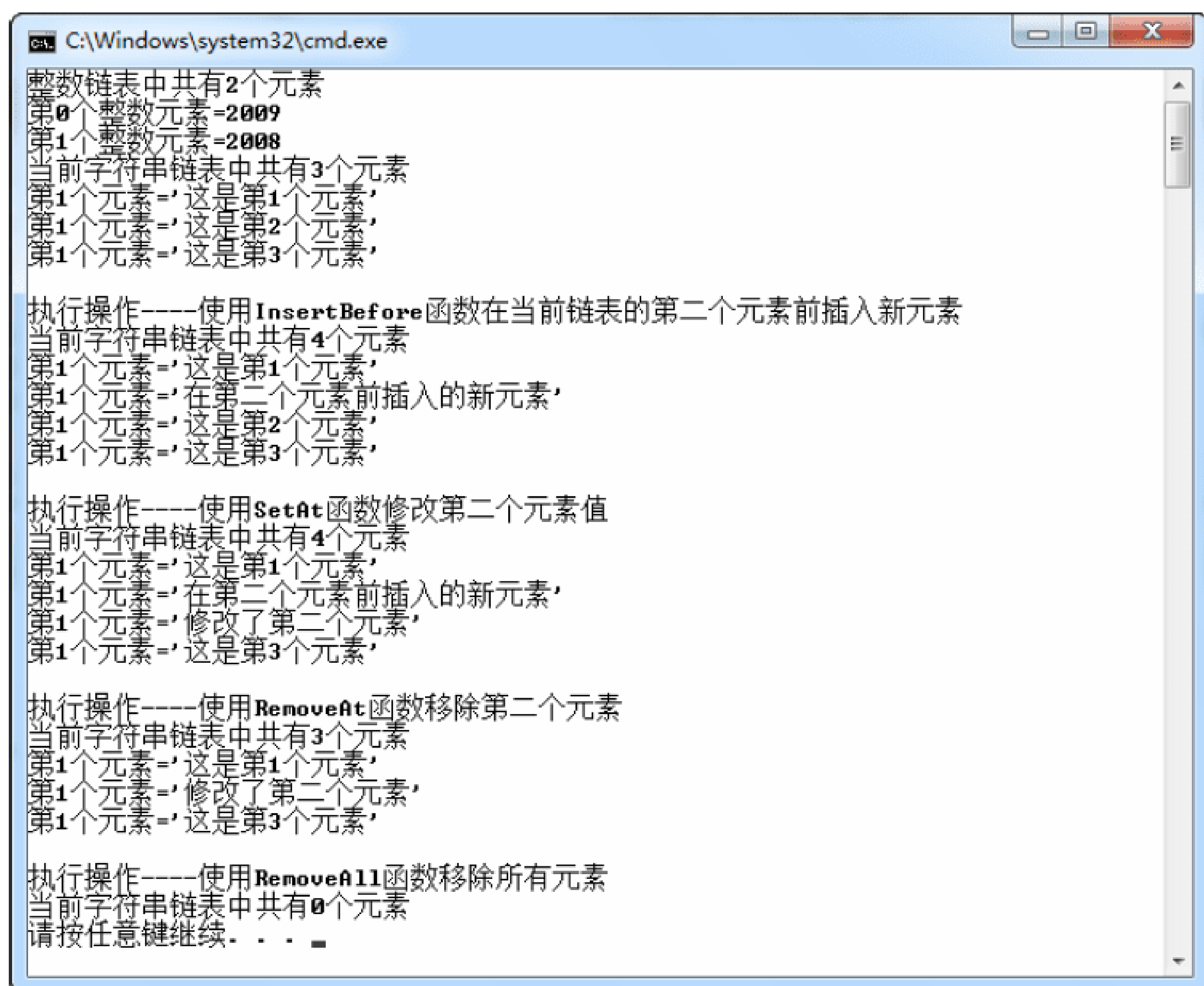


图 8-3　链表类使用实例运行效果

8.3　日期、时间类

日期、时间类是专门用于表示时间概念的。MFC 提供了强大的日期时间类——CTime 类，使用它可以完成常见的有关日期和时间的操作。在实际情况中，经常会遇到定时任务，因此，本节的最后引入了计时器的使用。

8.3.1　CTime 类

CTime 类没有基类，表示一个绝对时间和日期，是一个 time_t 的数据类型，并且与运行时库函数相连，包括与格林威治日期和 24 小时制时间之间的转换。CTime 的值是基于 UTC 时区的，格林威治时间（GMT）是零时区，而中国大部分是跨越 6 时区到 8 时区的。计算机本地的时区是在 TZ 的环境变量中设置的。

CTime 对象的大小是 4 个字节，不能派生而且大部分函数是内联函数。表 8-4 列出了 CTime 类的主要成员函数。

表 8-4　CTime类的成员函数

函　数　名	功　　能
GetCurrentTime()	创建一个取值为当前时间的 CTime 对象。此函数为静态函数
GetTime()	返回 CTime 对象对应的 time_t 类型的值
GetYear()	返回 CTime 对象当前时间的年的取值
GetMonth()	返回 CTime 对象当前时间的月的取值，范围为 1～12
GetDay()	返回 CTime 对象当前时间的日的取值，范围为 1～31
GetHour()	返回 CTime 对象当前时间的小时的取值，范围为 0～23
GetMinute()	返回 CTime 对象当前时间的分钟的取值，范围为 0～59
GetSecond()	返回 CTime 对象当前时间的秒的取值，范围为 0～59
GetDayOfWeek()	返回 CTime 对象当前时间是星期几，其中 1 代表星期日，2 代表星期一，依次类推
GetGmtTm()	将 CTime 对象的时间按照 UTC 解释，并将其赋值给 tm 结构
GetLocalTm()	将 CTime 对象的时间按照本地时区解释，并将其赋值给 tm 结构
GetAsSystemTime()	将 CTime 对象转换成 Win32 兼容的 SYSTEMTIME 结构
Format()	将 CTime 对象转换成基于本地时区的格式化字符串
FormatGmt()	将 CTime 对象转换成基于格林威治时间的格式化字符串
<<	对象序列化操作符，将对象输出到 CArchive 对象或 CdumpContext 对象中
>>	对象反序列化操作符，从 CArchive 对象导入 CTime 对象
=	为 CTime 对象赋值
+、–	CTime 对象与 CTimeSpan 对象或 CTime 对象之间进行相加和相减的运算
+=、–=	赋值加减运算，操作与+、–运算相同，区别在于这两个操作数会将运算结果赋值给目标对象，即操作符左边的对象
==、<、etc	比较 CTime 对象的绝对时间

8.3.2　格式化 CTime 对象

格式化 CTime 对象有两种方法，一种是通过构造函数，一种是通过 Format()成员函数。构造函数方法通常用于与多种时间结构之间进行转换。代码如下：

```
CTime( );                        //不带参数的构造函数
CTime( const CTime& timeSrc );  //带 CTime 参数的构造函数
CTime( time_t time );            //带 time_t 参数的构造函数
//带年、月、日、时、分、秒参数的构造函数
CTime( int nYear, int nMonth, int nDay, int nHour, int nMin,
                           int nSec, int nDST = -1 );
CTime( WORD wDosDate, WORD wDosTime, int nDST = -1 );
                                 //带日期值、时间值和差值的构造函数
CTime( const SYSTEMTIME& sysTime, int nDST = -1 );
                                 //带 SYSTEMTIME 参数的构造函数
CTime( const FILETIME& fileTime, int nDST = -1 );
                                 //带 FILETIME 参数的构造函数
```

上面代码中的所有构造函数都会创建一个 CTime 对象，并使用当前时区的指定的绝对时间初始化对象。timeSrc 参数表示已经存在的 CTime 对象，使用此参数初始化 CTime 对象，会将此对象的值赋值给新建的对象。time 参数表示一个 time_t 类型的时间值，使用此参数初始化 CTime 对象，会将此对象的值赋值给新建的 CTime 对象。参数 nYear、nMonth、nDay、nHour、nMin、nSec，分别表示初始化 CTime 对象的年、月、日、时、分、秒。参数 wDosDate 和 wDosTime 表示 MS-DOC 的日期和时间值转换成 CTime 对象。参数 sysTime 表示使用 SYSTEMTIME 结构初始化 CTime 对象。参数 fileTime 表示使用 FILETIME 的结构值初始化 CTime 对象。而 nDST 参数表示是否使用夏时制，0 表示使用标准时间，大于 0 表示使用夏时制，小于 0 则根据计算机上的设置决定是否使用夏时制。具体使用如下代码：

```
time_t osTime;                   //定义时间结构
time( & osTime) ;                //获取当前操作系统的时间
CTime time1;                     //空 CTime
CTime time2 = time1;             //CTime 的赋值操作
CTime time3(osTime);             //从 time_t 结构构造 Ctime
//2009 年 1 月 19 日，22 点 15 分 59 秒
CTime time4( 2009, 1, 19, 22, 15, 59 );
```

除了初始化 CTime 对象外，还可以使用 Format()函数格式化 CTime 对象。其函数原型为：

```
CString Format( LPCTSTR pFormat ) const;    //使用指定格式格式化字符串
CString Format( UINT nFormatID ) const;     //使用格式化 ID 格式化字符串
```

格式化函数的返回值为 CString，其中包含格式化后的时间值。而参数 pFormat 与 printf 格式化字符串是类似的，百分号后加上格式化代码。参数 nFormatID 表示格式化标识符字符串的 ID。有效的格式化代码如下。

- %D：CTime 时间的当前天数。

- %H：当天的小时数。
- %M：当前小时的分钟数。
- %S：当前分钟的秒数。
- %%：百分号。

此函数会创建代表日期/时间值的格式化字符串，但是如果 CTime 对象为 NULL 或其值无效，则返回的字符串为空字符串。具体例子如下：

```
CTime t( 2009, 1, 19, 22, 15, 59 );        //使用年月日时分秒初始化 CTime 值
CString s = t.Format( "%A, %B %d, %Y" );   //格式化字符串
//判断格式化后的时间字符串与指定值是否相同
ASSERT( s == "Friday, January 19, 2009" );
```

8.3.3　CTimeSpan 类

与 CTime 一起使用的类还有 CTimeSpan 类，表示一个时间间隔，即两个不同的 CTime 对象之间的间隔。CTimeSpan 也没有基类。与 CTime 类不同的是，它代表一个相对时间间隔，也是结合 ANSI 的 time_t 数据类型，并且与运行时函数相关。CTimeSpan 对象以秒保留时间。因为它存储为 4 个字节的有符号数，因此，最大时间间隔值为±68 年。表 8-5 列出了 CTimeSpan 类的常用成员函数。

表 8-5　CTimeSpan类的成员函数

函　数　名	功　　能
GetDays()	返回 CTimeSpan 对象中的整天的数目
GetHours()	返回 CTimeSpan 对象中的当天的小时数
GetTotalHours()	返回 CTimeSpan 对象中的整小时数
GetMinutes()	返回 CTimeSpan 对象中的当前小时的分钟数
GetTotalMinutes()	返回 CTimeSpan 对象中的整分钟数
GetSeconds()	返回 CTimeSpan 对象中的当前分钟的秒数
GetTotalSeconds()	返回 CTimeSpan 对象中的整秒数

下面的代码示例演示了 CTimeSpan 对象的使用。

```
CTimeSpan ts( 7, 5, 5, 1 );             //初始化 CTimeSpan
ASSERT( ts.GetDays() == 7 );            //判断 CTimeSpan 对象的天数是否为 7
ASSERT( ts.GetHours() == 5 );           //判断 CTimeSpan 对象的小时数是否为 5
ASSERT( ts.GetMinutes () == 5 );        //判断 CTimeSpan 对象的分钟数是否为 5
ASSERT( ts.GetSeconds () == 1 );        //判断 CTimeSpan 对象的秒数是否为 1
```

上面的代码首先定义了 CTimeSpan 对象，并为其初始化值。然后分别使用 ASSERT 语句从 CTimeSpan 对象中获取日、时、分、秒并与预期的值进行比较。

8.3.4　制作一个计时器

使用 CTime 类和 CTimeSpan 类可以实现计时器的功能。在启动计时器时，记录当前时间到 m_BeginTime 变量中，并启动定时器。在定时器处理函数中，记录结束时间到 m_EndTime 变量中，然后计算 m_BeginTime 变量和 m_EndTime 变量之间的差值存储在

CTimeSpan 类型的变量中，最后使用 GetSeconds()函数计算出时间差的秒数，显示在界面上。代码如下：

```
01  void CTimerSampleDlg::OnButtonStart()
02  {
03      SetTimer(100, 1000, NULL);
04      m BeginTime = CTime::GetCurrentTime();
05  }
06  void CTimerSampleDlg::OnButtonStop()
07  {
08      KillTimer(100);
09  }
10  void CTimerSampleDlg::OnTimer(UINT nIDEvent)
11  {
12      CTime m EndTime = CTime::GetCurrentTime();
13      CTimeSpan m Span = m EndTime - m BeginTime;
14      CString log;
15      log.Format("%d", m Span.GetSeconds());
16      m StaticTime = log;
17      UpdateData(FALSE);
18      CDialog::OnTimer(nIDEvent);
19  }
```

上面的代码首先调用 CTime::GetCurrentTime()函数获取当前时间，然后将其与开始时间相减，赋值到 CTimeSpan 对象中，并在界面上显示两个时间相差的秒数。程序运行效果如图 8-4 所示。

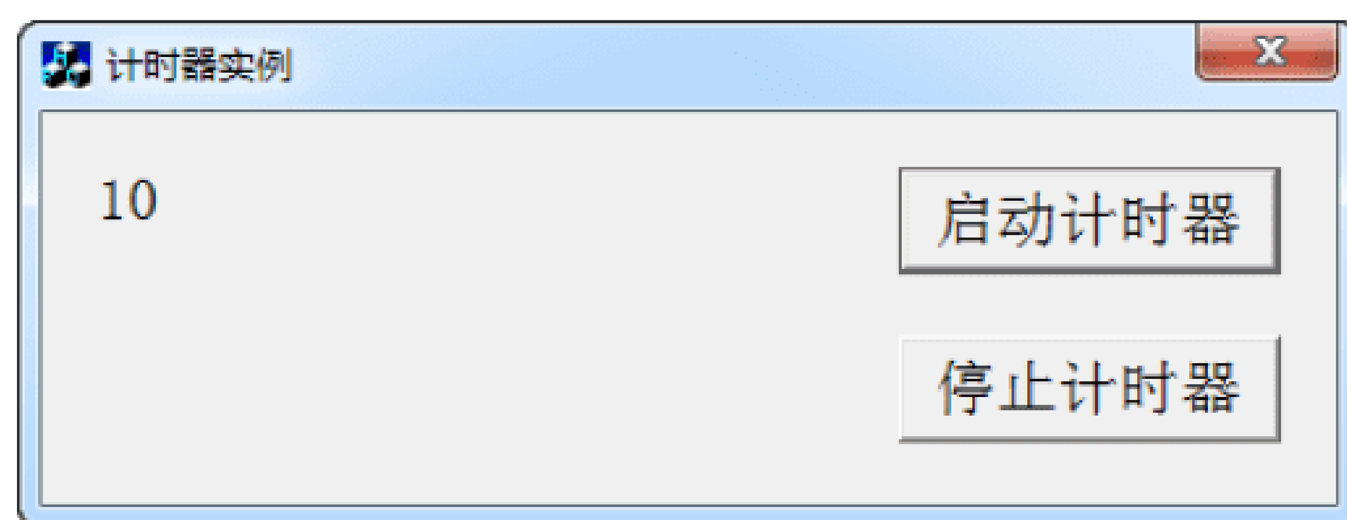


图 8-4　计时器实例运行效果

8.4　MFC 文件操作类——CFile

MFC 中使用 CFile 类封装了有关文件的操作，它是 MFC 的基类，并直接提供了非缓冲的、二进制磁盘输入/输出服务。通过派生类间接支持文本文件和内存文件，并且与 CArchive 类结合使用可以提供对序列化的支持。支持的文件操作包括文件的创建、打开、读文件、写文件以及文件定位等操作。本节将介绍有关 CFile 的这些操作，并在后面以一个实例说明 CFile 的使用方法。

8.4.1　构造文件对象并打开文件

要操作文件对象 CFile，首先要构造文件对象并打开文件。CFile 类的构造方式有以下

3 种。

```
CFile( );                          //不带参数的构造函数
CFile( int hFile );                //带文件句柄参数的构造函数
//带文件名和选项参数的构造函数
CFile(
    //要打开的文件路径，此路径可以是绝对路径，也可以是相对路径
    LPCTSTR lpszFileName,
    UINT nOpenFlags                //打开文件时的共享和访问模式
);
//打开文件
virtual BOOL Open(
    //要打开的文件路径，此路径可以是绝对路径，也可以是相对路径
    LPCTSTR lpszFileName,
    UINT nOpenFlags,               //打开文件时的共享和访问模式
    //打开文件的异常捕获变量
    CFileException* pError = NULL
);
```

在上面的 4 个函数中，前 3 个是构造函数，第 4 条语句是打开文件的函数。其中，lnOpenFlags 参数指定共享和访问模式。它可以是下列选项中的任意组合，其中使用位或(|)组合。

- ❑ CFile::modeCreate：新建文件，如果文件已经存在，则截取文件的长度为 0。
- ❑ CFile::modeNoTruncate：与 modeCreate 组合使用。如果创建的文件已经存在，则不会截取为 0 长度。因此，其含义是打开已经存在的文件或新建文件。
- ❑ CFile::modeRead：以只读方式打开文件。
- ❑ CFile::modeReadWrite：以可读写方式打开文件。
- ❑ CFile::modeWrite：以只写方式打开文件。
- ❑ CFile::modeNoInherit：阻止文件从子进程中继承。
- ❑ CFile::shareDenyNone：共享读写的打开文件。
- ❑ CFile::shareDenyRead：排它读权限打开文件。
- ❑ CFile::shareDenyWrite：排它写权限打开文件。
- ❑ CFile::shareExclusive：排它模式打开文件。
- ❑ CFile::typeText：文本模式打开文件。
- ❑ CFile::typeBinary：二进制模式打开文件。

以下代码显示了创建文件，并以写模式打开该文件。

```
01  CString filename = "comm.log";               //定义文件名变量
02  TRY
03  {
04      //以创建和可写方式打开文件
05      CFile f(filename, CFile::modeCreate | CFile::modeWrite );
06  }
07  CATCH( CFileException, e )                   //打开文件发生异常时
08  {
09     #ifdef _DEBUG                              //如果当前是调试模式
10      //显示打开文件错误的原因
11      afxDump << "打开文件失败" << e->m_cause << "\n";
12     #endif
13  }
14  END_CATCH
```

上面代码使用构造函数打开 comm.log 文件，如果文件不存在，则创建文件，并且允许向文件中写入数据。

8.4.2　读写文件

CFile 类提供了下面两个函数实现读文件的功能：

```
virtual UINT Read(            //从文件中获取指定长度的数据到缓冲区中
       void* lpBuf,           //存放读取的数据缓冲区指针
       UINT nCount );         //要读取的字节数
DWORD ReadHuge(               //获取指定长度的大量数据
       void* lpBuffer,        //存放读取的数据缓冲区指针
       DWORD dwCount );       //要读取的字节数
```

上面这两个函数的功能都是从文件中读取数据，区别在于 Read()函数最多可以读取 64K–1 个字节，而超过 64K 的数据，则需要使用 ReadHuge()函数读取。这两个函数的返回值是读取的字节数。下面是使用 Read()函数从文件中读取数据的代码。

```
01  char pbuf[50];                                //定义字符数组
02  UINT nBytesRead = cfile.Read( pbuf, 50 );     //从文件变量中读取 50 个字符
```

CFile 类提供了下面两个函数来实现写文件的功能。

```
virtual void Write(           //向文件中写入指定长度的数据
       const void* lpBuf,     //存放写入文件的数据缓冲区指针
       UINT nCount );         //要写入的字节数
void WriteHuge(               //向文件中写入指定长度的大量数据
       const void* lpBuf,     //存放写入文件的数据缓冲区指针
       DWORD dwCount );       //要写入的字节数
```

上面这两个函数的功能都是向文件中写入数据，区别在于，Write()函数一次最多可以写入小于 64K–1 个字节的数据，而超过 64K 的数据，则需要使用 WriteHuge()函数写入。下面是使用 Write()函数向文件中写数据的代码。

```
01  char pbuf[100];               //定义字符数组
02  cfile.Write( pbuf, 100 );     //向文件中写入 100 个字符
```

CFile 类还有一个 Flush()函数，作用是强制将文件缓冲区中的内容刷新到文件中。

```
virtual void Flush( );           //刷新文件缓冲区中的数据
```

8.4.3　定位文件

CFile 类提供了几个在文件中定位的函数，使用这些函数可以实现在文件中的快速定位。其函数原型如下：

```
virtual LONG Seek( LONG lOff, UINT nFrom );     //定位文件中指定位置的数据
void SeekToBegin( );                            //定位到文件头
DWORD SeekToEnd( );                             //定位到文件尾
virtual DWORD GetLength( ) const;               //获取文件的长度
virtual void SetLength( DWORD dwNewLen );       //设置文件的长度
```

其中，Seek()函数按照条件执行文件定位。lOff 参数表示要定位的字符的偏移量。nFrom 参数用于指定位模式，有下面 3 种模式。

- CFile::begin：从文件头开始定位字符。
- CFile::current：从文件当前位置开始定位字符。
- CFile::end：从文件尾开始定位字符。

如果定位成功，则 Seek()函数返回定位位置的字符；否则，函数抛出异常。

SeekToBegin()函数定位到文件头。SeekToEnd()函数定位到文件尾，返回值为文件的字节数。GetLength()函数返回文件的长度。SetLength()函数会扩展或截取文件的长度，其中 dwNewLen 是要设置的文件的长度。

以下代码演示了这几个定位函数的使用方法。

```
01  LONG lOff = 225, lResult;                    //定义文件定位使用的变量
02  //定位到文件的第 226 个字符的位置
03  lResult = cfile.Seek( lOff, CFile::begin );
04  cfile.SeekToBegin();                         //定位到文件头
05  DWORD dwResult = cfile.SeekToEnd();          //定位到文件尾
06  DWORD dwNewLen = 100;                        //定义使用的长度变量
07  cfile.SetLength(dwNewLen);                   //设置文件的长度为 100
```

8.4.4　文件管理操作

除了基本的读写和定位函数外，CFile 还提供了一些对文件的管理操作函数，如表 8-6 所示。

表 8-6　文件管理函数

函　数　名	功　　能
LockRange()	锁定文件中的一个范围的字节
UnlockRange()	解锁文件中的一个范围的字节
GetPosition()	获取文件当前的位置指针
GetStatus()	获取打开文件的状态
GetFileName()	获取选定的文件名
GetFileTitle()	获取选定的文件标题
GetFilePath()	获取选定的文件的完整路径
SetFilePath()	设置选定的文件的完整路径
Rename()	重命名指定文件
Remove()	删除指定文件
GetStatus()	获取指定文件状态
SetStatus()	设置指定文件状态

下面的代码演示了这些函数的使用方法。

```
01  DWORD dwPos = 5;                    //定义位置变量
02  DWORD dwCount = 20;                 //定义个数变量
03  //锁定指定位置处，指定个数的文件的内容
04  cfile.LockRange( dwPos, dwCount );
```

```
05  ...                                              //处理获取的数据
06  cfile.UnlockRange( dwPos, dwCount );             //解锁对文件的锁定
07  DWORD dwPos = cfile.GetPosition();               //获取文件当前的位置
08  CFileStatus status;                              //定义文件状态变量
09  cfile.GetStatus( status );                       //获取当前文件的状态
10  char* pOldName ="old.log";                       //定义原来的文件名变量
11  char* pNewName = "new.log";                      //定义新的文件名变量
12  CFile::Rename( pOldName, pNewName );             //重命名文件名
13  char* pFileName = "test.log";                    //定义文件名变量
14  try
15  {
16      CFile::Remove( pFileName );                  //删除指定文件
17  }
18  catch( CFileException, e )                       //如果删除文件时发生异常
19  {
20      #ifdef _DEBUG                                //如果当前是调试版本
21          //显示提示信息
22          afxDump << "删除文件 " << pFileName << "失败 \n";
23      #endif
24  }
25  BYTE newAttribute;                               //定义属性字节
26  CFileStatus status;                              //定义文件状态变量
27  CFile::GetStatus( pFileName, status );           //获取文件状态
28  status.m_attribute = newAttribute;               //设置文件状态加入定义的属性
29  CFile::SetStatus( pFileName, status );           //设置文件状态
```

上面的代码定义了 CFile 对象，演示了数据锁定和数据解锁的方法，并演示了如何使用文件状态对象获取和设置文件属性的方法。

8.4.5　文件操作实例

使用 CFile 类可以完成对文件的所有操作。本小节使用 CFile 类实现写文件和读文件的功能。在对 CFile 对象进行操作前，首先需要调用 Open()函数打开文件，然后调用 Write()函数或 Read()函数进行写操作或读操作，最后调用 Close()函数关闭 CFile 对象。代码如下：

```
01  #define MAX_FILE_LEN       250
02  void CFileSampleDlg::OnButtonRead()              //读文件
03  {
04      CFile file;                                  //定义 CFile 变量
05      file.Open("data.txt", CFile::modeRead);      //打开文件
06      //获取要读取的内容的长度
07      int len = min(file.GetLength(), MAX_FILE_LEN);
08      char buf[MAX_FILE_LEN]={0};                  //定义存放内容的缓冲区
09      file.Read(buf, len);                         //从文件中读取文件内容
10      m_Content.Format("%s", buf);                 //赋值给内容提示框
11      file.Close();                                //关闭文件
12      UpdateData(FALSE);                           //刷新显示
13  }
14  void CFileSampleDlg::OnButtonWrite()             //写文件
15  {
16      UpdateData();                                //从内容框中获取内容
17      CFile file;                                  //定义 CFile 变量
```

```
18      //打开文件
19      file.Open("data.txt", CFile::modeCreate | CFile::modeWrite);
20      //向文件中写入内容
21      file.Write(m_Content, m_Content.GetLength());
22      file.Close();                                //关闭文件
23  }
```

在上面的代码中，OnButtonRead()函数实现了读文件的功能。它定义了 CFile 对象，调用 CFile 对象的 Open()函数打开指定文件，并计算读取数据的最短长度。使用 CFile 对象的 Read()函数从文件中读取数据，并格式化显示在控件中，最后关闭 CFile 对象。OnButtonWrite()函数实现了写文件的功能。首先调用 CFile 对象的 Open()函数，然后调用 Write()函数向文件中写入数据，最后调用 CFile 对象的 Close()函数关闭文件。程序运行的效果如图 8-5 所示。

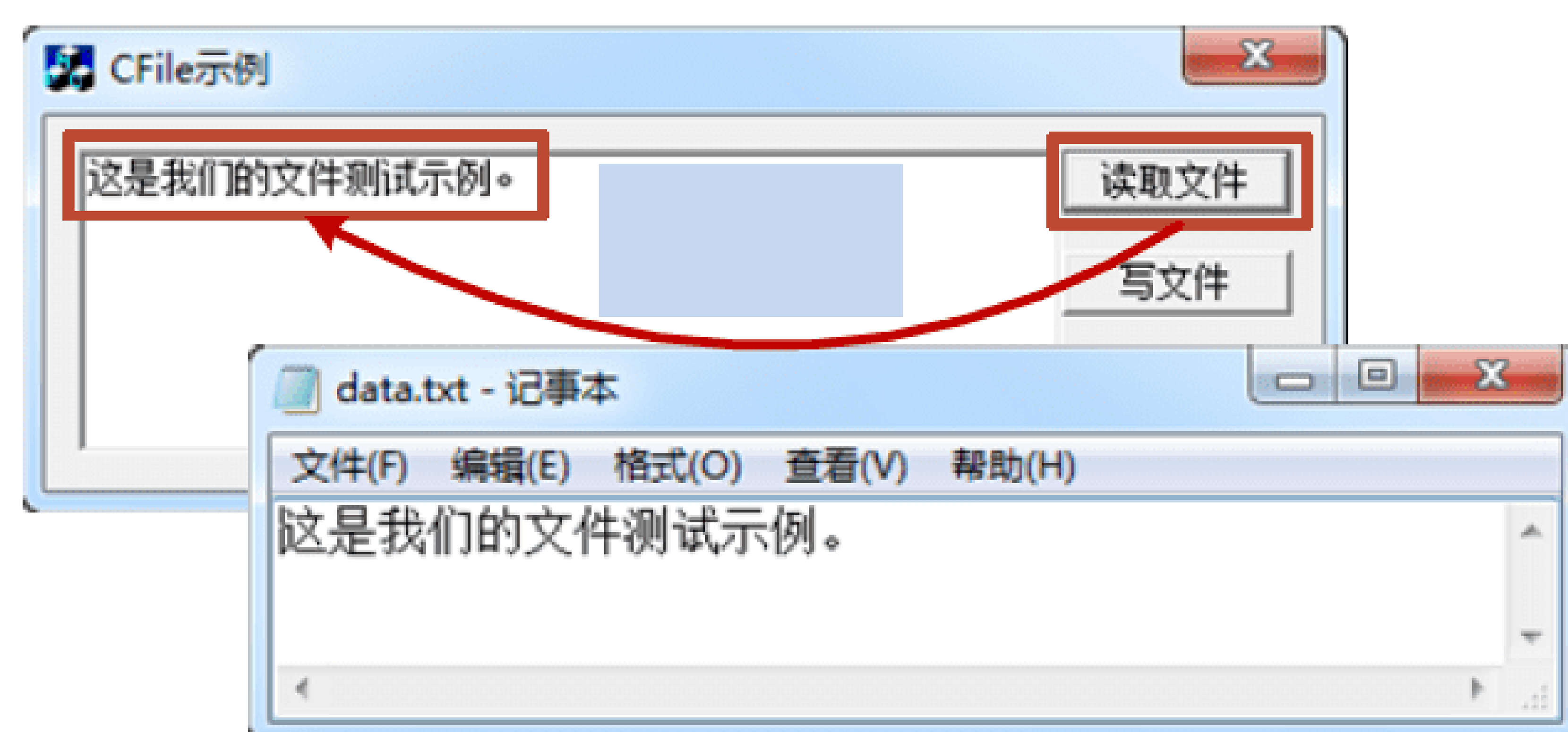


图 8-5　文件操作实例运行效果

8.5　MFC 异常类

MFC 为异常提供了强大的支持，其中 CException 类是 MFC 中的所有异常类的基类。使用 MFC 的异常捕获方式，可以增强程序的健壮性，并能为用户提供较详细的错误提示。本节将以 CFileException 类为例介绍 MFC 中支持的异常类，并演示捕获异常的方法。

8.5.1　MFC 异常类简介

Cexception 是 MFC 异常类的基类，封装了所有异常共有的操作。使用 Cexception 类的 GetErrorMessage()成员函数可以获取描述异常的说明。使用 ReportError()成员函数可以以消息框的方式向用户报告错误消息。在 Cexception 类的基础上，MFC 提供了对应多种操作的异常类，使程序可以准确定位异常情况。表 8-7 中列出了 MFC 中支持的异常类，这些类全部继承自 Cexception 类。

表 8-7　MFC异常类

异　常　类	实现的功能
CmemoryException	管理内存异常
CnotSupportedException	管理不支持的请求异常

续表

异　常　类	实现的功能
CArchiveException	管理序列化异常
CFileException	管理文件异常
CResourceException	管理资源异常，主要是资源没有找到或创建失败的情况
COleException	OLE 异常
CDBException	数据库异常，主要是基于 MFC 的 ODBC 操作发生的异常
COleDispatchException	OLE 调度异常
CUserException	表示资源找不到的异常
CDaoException	数据访问对象异常，由 DAO 类产生
CInternetException	Internet 异常

8.5.2　文件异常类 CFileException

CFileException 类表示与文件相关的异常情况，它在 CFile 的成员函数和 CFile 派生类的成员函数中构造并抛出。用户可以在 CATCH 代码段中访问 CFileException 对象。通常通过原因代码可以获得异常的原因。CFileException 类包含下面 3 个数据成员。

- m_cause：包含异常对应的原因代码。
- m_lOsError：包含与操作系统相关的错误代码。
- m_strFileName：包含异常对应的文件名。

其中，m_cause 是用于表示文件异常的直接原因的，有效取值如下所述。

- CFileException::none：没有错误发生。
- CFileException::generic：发生未指定的错误。
- CFileException::fileNotFound：没有找到文件。
- CFileException::badPath：文件路径无效。
- CFileException::tooManyOpenFiles：打开的文件太多。
- CFileException::accessDenied：文件不能访问。
- CFileException::invalidFile：无效的文件句柄。
- CFileException::removeCurrentDir：要移除的文件夹是当前工作的文件夹，不能移除。
- CFileException::directoryFull：文件夹已满。
- CFileException::badSeek：在定位文件时发生错误。
- CFileException::hardIO：发生硬件错误。
- CFileException::sharingViolation：没有装载 SHARE.EXE，或已锁定了共享区域。
- CFileException::lockViolation：要锁定的区域已经被锁定了。
- CFileException::diskFull：磁盘已满。
- CFileException::endOfFile：已到达文件尾。

8.5.3　异常的捕获

程序在运行过程中，发生异常会抛出异常对象，其中包含与异常相关的信息。而要捕获异常，需要使用 THROW、THROW_LAST、TRY、CATCH、AND_CATCH 和 END_CATCH

宏。要捕获特殊的异常，则必须使用 CException 类相应的派生类。要捕获所有类型的异常，可以使用 CException 类，然后使用 CObject::IsKindOf()函数判断是否是 CException 类的派生类的对象。可以调用 GetErrorMessage()函数或 ReportError()函数报告异常的详细情况。这两个函数在 CException 的所有派生类中都可以使用。

要注意的是，CException 类是抽象基类，因此，不能创建 CException 对象，要抛出异常，必须创建 CException 类的派生类对应的对象。如下代码，显示了在打开文件时，如果没有找到文件，则捕获文件异常的处理情况。用户可以根据自己的需要，扩充对异常处理的支持。

```
01  char* pFileName="omm.ini";                    //定位文件名变量
02  TRY
03  {
04     CFile f( pFileName, CFile::modeRead );    //可写的打开文件
05  }
06  //如果打开文件发生异常，则处理异常
07  CATCH( CFileException, e )
08  {
09       //如果发生异常时文件未找到，则打印错误原因
10      if( e->m_cause == CFileException::fileNotFound )
11          printf( "错误：文件没有找到\n");
12  }
13  END_CATCH
```

上面的代码中，首先定了存放文件名的变量，然后使用 CFile 打开文件，在此过程中，使用 TRY、CATCH 和 END_CATCH 捕获打开文件时发生的异常，并判断异常是否为 CFileException::fielNotFound()函数，如果是，则向屏幕输出错误原因。程序运行效果如图 8-6 所示。

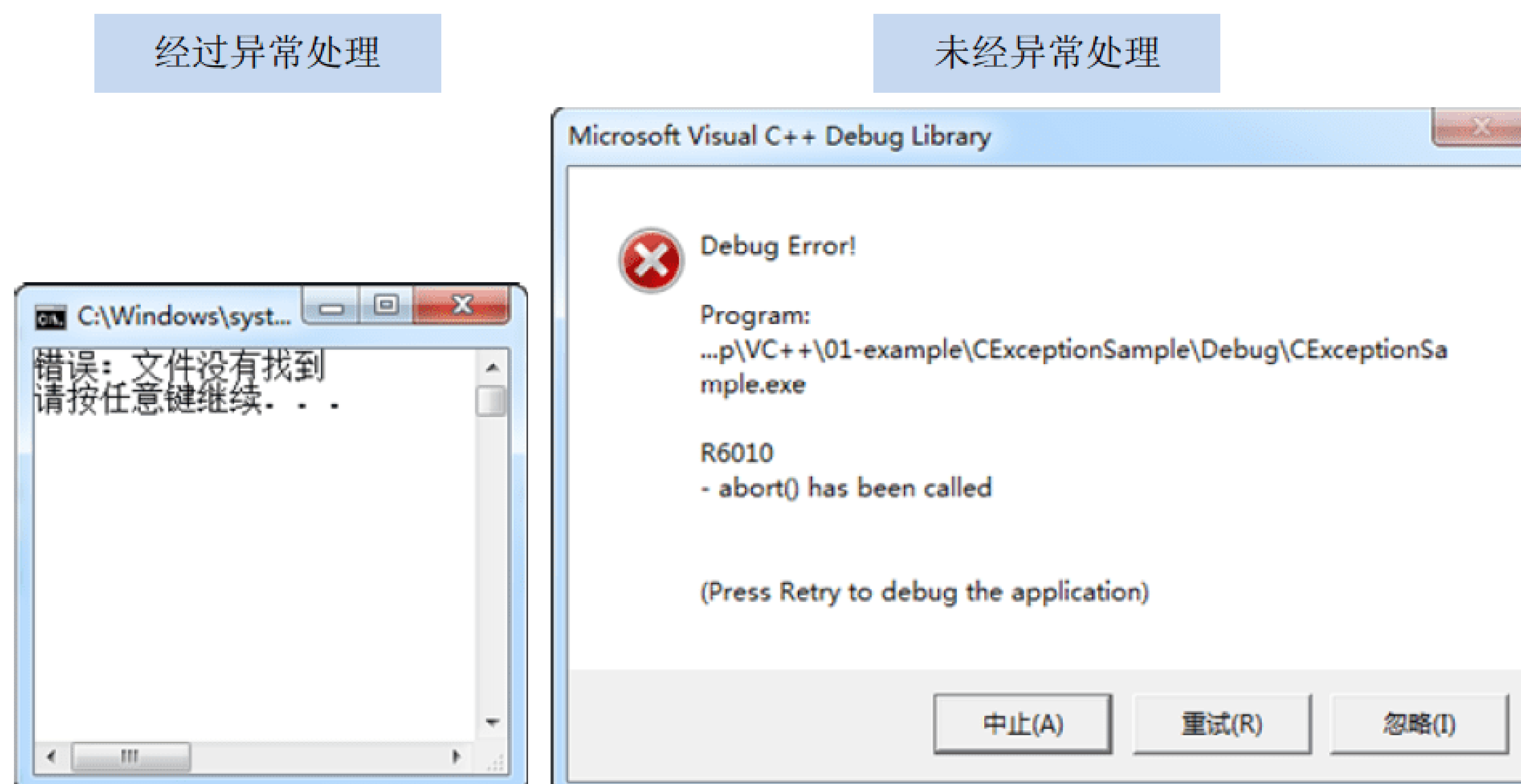


图 8-6　异常捕获运行效果

在图 8-6 中，左图是进行异常处理后输出的异常信息。右图是没有进行异常处理，程序输出的异常信息。由此可以看出，准确地处理各种异常情况对于构建健壮的程序来说是非常重要的。

8.6　本 章 小 结

本章介绍了 MFC 中的几个常用类。重点介绍了字符串类 CString 类、数组类 CArray 类、链表类 CList 类、日期时间类 CTime 类、时间间隔类 CTimeSpan 类、文件类 CFile 和异常类 CException，并通过实例讲解了这些类的使用方法。熟练使用这些类的各种用法是进行后面程序开发的基础。第 9 章将讲解 VC 中比较重要的程序结构——文档/视图结构。

8.7　习　　题

1．模仿 8.1.5 小节的实例，处理字符串“What you name?”：

（1）获取字符串中的“you”。

（2）计算字符串的长度。

（3）在“What”和“you”中间添加“is”。

【思路】参照 8.1.5 小节的实例，本题调用的函数是一样的。

2．模仿 8.2.2 小节的实例，构造一个 int 型的数组对象，如下：

```
CArray<int,int> intArray;
```

完成下列操作。

（1）添加 3 个数据：100、1000 和 10000。

（2）修改第 2 个数据为 5000。

（3）删除前两个数据，即数据 100 和 5000。

（4）计算数组中的对象数目。

【思路】参照 8.2.2 小节的实例。

3．创建基于对话框的应用程序 Dlg，在对话框上添加 4 个控件：两个编辑框和两个按钮，如图 8-7 所示。实现的功能是：单击“保存内容到文件” 按钮，可以保存左上角编辑框中的文本；单击“显示内容到编辑框”按钮，可以读取文件的内容，然后显示在右下角的编辑框之中。



图 8-7　习题 3 对话框界面设计

【思路】CFile 类提供了 4 个成员函数可以方便地完成此题所要求的功能。这 4 个函数是：Open()、Write()、Read()和 Close()。

第 9 章　文档/视图结构应用程序

本章将讲解有关文档/视图结构及其高级应用，如序列化、多文档应用程序开发和对话框的分割及多视等技术。

9.1　文档/视图结构分析

本节将深入分析文档/视图结构的框架及其实现核心。

9.1.1　框架中的主要类

MFC 通过多个类提供了对程序框架的支持，使用这些类可以简单地实现文档/视图结构。其中主要包括以下 5 个类。这些类之间的关系如图 9-1 所示。

- ❑ 应用程序类（CWinApp）：是 MFC 程序的应用程序管理类，也是程序的入口类。
- ❑ 文档模板类（CDocTemplate）：用于管理应用程序的一组文档视图和框架。
- ❑ 框架类（CMainFrame）：用于管理 Windows 对话框类，框架对话框会保存当前视图的指针，当其他视图被激活时，指针会随时更新。
- ❑ 文档类（CDocument）：保存用户数据，提供用户定义的文档类的基本功能。文档表示用户使用 File|Open 命令打开和使用 File|Save 命令保存的数据的单位。CDocument 类提供标准操作，如创建文档、装载文档和保存文档。框架使用 CDocument 定义的接口操作文档。
- ❑ 视图类（CView）：主要完成数据的显示功能。

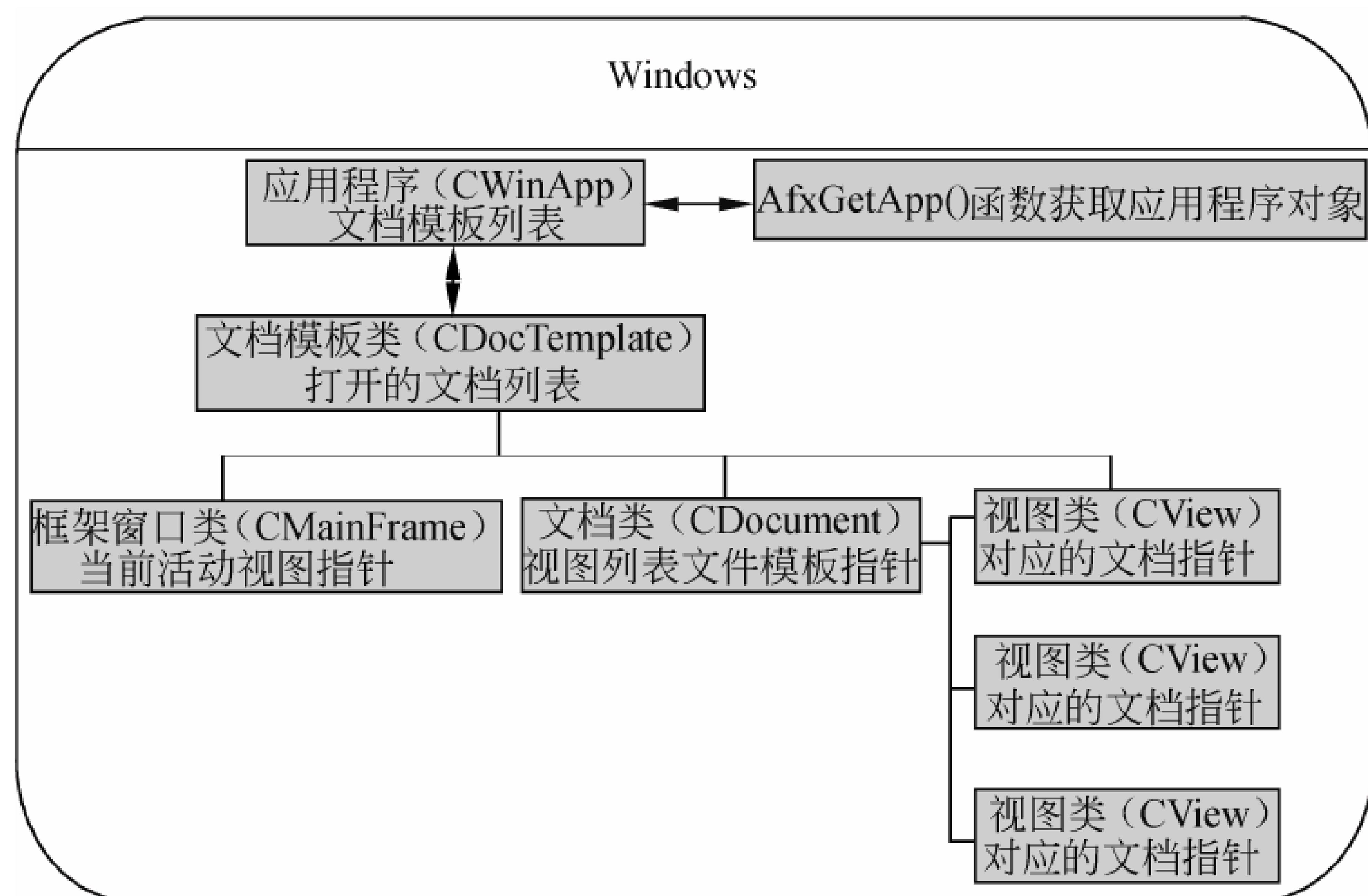


图 9-1　文档/视图框架中的主要类及其关系

从图 9-1 中可以看出，应用程序对象 CWinApp 类包含文档模板列表，使用多个文档模板可以支持多种类型的文档。每个文档模板类 CDocTemplate 包含打开的文档列表，并且每个文档模板有与其对应的框架窗口类 CMainFrame、文档类 CDocument 和视图类 CView。框架对话框包含当前活动视图的指针。文档包含与其相关的视图列表和创建文档的文档模板指针。视图包含与其相关的文档的指针，并且是父框架对话框的子对话框。Windows 保留所有打开的对话框，并且可以向视图发送消息。表 9-1 列出了文档/视图结构中的各个对象中可以访问的其他对象。

表 9-1　文档/视图结构中的各个对象可以访问的其他对象

对　　象	可以访问的对象
文档	调用 GetFirstViewPosition()函数和 GetNextView()函数可以访问文档的视图列表。调用 GetDocTemplate()函数可以获取文档模板
视图	调用 GetDocument()函数可以获得与其相关的文档。调用 GetParentFrame()函数可以获得框架对话框
文档框架对话框	调用 GetActiveView()函数可以获得当前视图。调用 GetActiveDocument()函数可以获得与当前视图相连的文档
MDI 框架对话框	调用 MDIGetActive()函数可以获得当前激活的 CMDIChildWnd 对象

除了这些访问关系外，MFC 类库还提供了下列全局函数用于访问 CWinApp 对象和其他全局信息。

- AfxGetApp()函数：获取 CWinApp 对象指针。
- AfxGetInstanceHandle()函数：获取当前应用程序实例句柄。
- AfxGetResourceHandle()函数：获取应用程序资源句柄。
- AfxGetAppName()函数：获取应用程序名称，等同于 CWinApp 对象的 m_pszExeName 成员变量。

CWinApp 类是派生 Windows 应用程序对象的基类。应用程序对象提供初始化应用程序实例的 InitInstance()函数和运行应用程序的 Run()函数。每个使用 MFC 的应用程序只能包含一个派生自 CWinApp 的对象。当程序运行时，首先构造此对象，并且当调用 WinMain()入口函数时，此对象就已经可用了。在 MFC 程序中，声明派生自 CWinApp 类的全局对象标识应用程序类。当派生 CWinApp 类时，可以重写 InitInstance()成员函数创建自己的应用程序主对话框对象。代码如下：

```
01  BOOL CMDISampleApp::InitInstance()
02  {
03      ...
04      CWinAppEx::InitInstance();
05      AfxEnableControlContainer();
06      EnableTaskbarInteraction(FALSE);
07
08      //使用 RichEdit 控件需要 AfxInitRichEdit2()函数
09      //AfxInitRichEdit2();
10
11      SetRegistryKey(_T("应用程序向导生成的本地应用程序"));
12      LoadStdProfileSettings(4);  //加载标准 INI 文件选项(包括 MRU)
13      ...
```

```
14      //注册应用程序的文档模板
15      //文档模板将用作文档、框架窗口和视图之间的连接
16      CMultiDocTemplate* pDocTemplate;
17      pDocTemplate = new CMultiDocTemplate(IDR_MDISampleTYPE,
18          RUNTIME_CLASS(CMDISampleDoc),
19          RUNTIME_CLASS(CChildFrame), //自定义 MDI 子框架
20          RUNTIME_CLASS(CMDISampleView));
21      if (!pDocTemplate)
22          return FALSE;
23      AddDocTemplate(pDocTemplate);
24
25      //创建主 MDI 框架窗口
26      CMainFrame* pMainFrame = new CMainFrame;
27      if (!pMainFrame || !pMainFrame->LoadFrame(IDR_MAINFRAME))
28      {
29          delete pMainFrame;
30          return FALSE;
31      }
32      m_pMainWnd = pMainFrame;
33      //仅当具有后缀时才调用 DragAcceptFiles
34      // 在 MDI 应用程序中，这应在设置 m_pMainWnd 之后立即发生
35
36      //分析标准 shell 命令、DDE 和打开文件操作的命令行
37      CCommandLineInfo cmdInfo;
38      ParseCommandLine(cmdInfo);
39
40      //调度在命令行中指定的命令。如果用/RegServer、/Register、/Unregserver
41      //或 /Unregister 启动应用程序，则返回 FALSE
42      if (!ProcessShellCommand(cmdInfo))
43          return FALSE;
44      //主窗口已初始化，因此显示它并对其进行更新
45      pMainFrame->ShowWindow(m_nCmdShow);
46      pMainFrame->UpdateWindow();
47
48      return TRUE;
49  }
```

上面代码是当使用 Visual Studio 2010 的应用向导创建多文档 MFC 程序时，向导自动生成的。其中，SetRegistryKey 语句用于设置注册表键。LoadStdProfileSettings 语句用于装载标准的 INI 选项文件。pDocTemplate 变量的定义及后面两条语句用于声明文档模板类，并将文档模板类增加到应用程序对象中。pMainFrame 变量定义主对话框框架类，并调用 ShowWindow()函数显示主窗口。

9.1.2　文档类、视图类核心函数

在 MFC 中，在主框架下可以通过文档类和视图类进行数据管理，CDocument 类提供文档类的基本功能。文档代表使用文件打开命令打开和文件保存命令保存的数据单元。如 Word 程序，打开和保存的数据单元是扩展名为 doc 的文件。框架使用 CDocument 类定义的接口操作文档数据，支持与文档相关的标准操作，如创建文档、装载数据和保存文档。

文档是框架标准命令程序的一部分，从标准用户接口组件或活动视图中接收命令，如 File|Menu 命令。如果文档类不为特定命令编写处理代码，则文档模板会将此命令进行默认处理。

应用程序可以同时支持一个或多个文档类型，如应用程序既可以支持 Excel 表格，也可以支持文本文档。每种文档类型都有与其关联的文档模板，文档模板指定文档类型使用的资源，如菜单、图标或快捷键等。

在文档/视图结构中，当文档数据被修改时，其对应的每个视图必须反映这些修改。CDocument 类提供 UpdateAllViews()成员函数通知视图文档内容有变化，因此视图可以根据需要重绘界面。同时，框架对象也会在关闭文件前提示用户保存数据内容。表 9-2 列出了 CDocument 类的核心函数。

表 9-2　CDocument类的核心函数

函　　数	功　　能
AddView()	附加视图到文档中
GetDocTemplate()	返回描述文档类型的文档模板指针
GetFirstViewPosition()	返回与文档相关的的第一个视图，可以用于枚举
GetNextView()	枚举与文档相关的视图列表
GetPathName()	返回与文档相关的数据文件的路径
GetTitle()	返回文档的标题
IsModified()	返回表示自从上次保存后，文档是否被修改过
RemoveView()	从文档中卸载一个视图
SetModifiedFlag()	设置标记，表示自从上次保存后，文档修改过
SetPathName()	设置文档使用的数据文件的路径
SetTitle()	设置文档的标题
UpdateAllViews()	通知与文档相关的所有视图，文档数据发生了变化
CanCloseFrame()	当关闭查看文档的框架对话框时调用此函数
DeleteContents()	当清除文档时调用此函数
OnChangedViewLis()t	当向文档中添加视图或移除视图时调用此函数
OnCloseDocument()	当关闭文档时调用的函数
OnNewDocument()	当创建新文档时调用的函数
OnOpenDocument()	当打开已经存在的文档时调用的函数
OnSaveDocument()	当保存文档到磁盘时调用的函数
ReportSaveLoadException()	当在打开或保存文档发生异常时调用此函数
GetFile()	返回 CFile 对象的指针
ReleaseFile()	释放文件，可以被其他应用程序调用
SaveModified()	重载函数，询问用户是否可以保存文档
PreCloseFrame()	框架对话框关闭前调用的函数
OnFileSendMail()	通过邮件消息发送文档
OnUpdateFileSendMail()	如果支持邮件功能，则打开发送邮件命令

在文档/视图结构中，通过 CView 视图类与文档交互。如视图可以在框架对话框中渲染一幅文档的图像，并解释用户的输入。一个文档可以有与其相连的多个视图。当用户打开文档的一个对话框，框架创建一个视图和与其相连的文档。文档模板指定用于显示每种

文档的视图和框架对话框类型。MFC 根据功能不同，提供了多种视图类。表 9-3 列出了 MFC 中从 CView 派生而来的视图类。

表 9-3　从CView派生而来的视图类

派生的视图类	功　　能
CView	所有视图的基类
CCtrlView	CTreeView、CTreeView、CListView、CEditView和 CRichEditView类的基类。这些类允许文档/结构使用Windows标准控件初始化
CEditView	基于Windows编辑控件的简单视图。允许输入和编辑文本，用于创建简单的文本编辑程序
CRichEditView	包含CRichEditCtrl对象的视图。与CEditView类似，主要区别在于此类处理格式化文本
CListView	包含CListCtrl对象的视图
CTreeView	包含CTreeCtrl对象的视图
CScrollView	CFormView、CRecordView和CDaoRecordView的基类，实现滚动视图内容的功能
CFormView	包含控件的对话框视图。基于对话框的应用程序提供过一个或多个这样的窗体接口
CHtmlView	Web浏览器视图，允许用户浏览WWW站点、本地文件系统和网络中的文件夹，也可以用于作为活动文档容器
CRecordView	在控件中显示ODBC数据记录的窗体视图。与CRowset相连
CDaoRecordView	在控件中显示DAO数据记录的窗体视图。与CDaoRecordset相连
COleDBRecordView	在控件中显示OLE DB数据记录的窗体视图。与CRowset相连

表 9-3 中的视图类除了 CView 外均是派生的视图类，除了 CCtrlView 和 CDaoRecordView 外，其余的都可以在 MFC 应用向导中使用。在向导中可以选择应用程序的视图类，在 Base Class 下拉菜单中选择合适的视图类。表 9-4 列出了 CView 类的核心函数。

表 9-4　CView类的核心函数

函　　数	功　　能
DoPreparePrinting()	显示打印对话框并创建打印机设备上下文，当重载 OnPreparePrinting()成员函数时，会调用此函数
GetDocument()	返回与视图相关的文档
OnInitialUpdate()	当第一次视图附加到文档上时，调用此函数
IsSelected()	检测文档项是否被选择
OnActivateView()	当激活视图时，调用此函数
OnActivateFrame()	当激活包含视图的框架对话框时，调用此函数
OnBeginPrinting()	当开始打印任务时，调用此函数
OnDraw()	在显示屏、打印或打印预览中渲染文档图像
OnEndPrinting()	当结束打印任务时，调用此函数
OnEndPrintPreview()	当退出打印预览时，调用此函数
OnPrepareDC()	当显示屏调用 OnDraw()成员函数前，以及为打印或打印预览调用 OnPrint()成员函数前调用此函数
OnPreparePrinting()	在打印或打印预览文档前，调用此函数
OnPrint()	打印或打印预览一页文档
OnUpdate()	通知视图文档内容已经修改

9.1.3　新建、保存和打开的实现

文档/视图结构的应用程序最主要的是要创建和管理文档。因此在使用文档前，首先需要创建文档。创建文档有如下两种方式：

- 使用 File|New 命令创建新的空文档，此时，在 CDocument 类的 OnNewDocument()重载函数中初始化文档。
- 使用 File|Open 命令打开文档并从文件中读取内容。此时，在 CDocument 类的 OnOpenDocument 重载函数中初始化文档。

如果这两种方式的初始化工作是相同的，则在这两个重载函数中调用一个公用的成员函数即可，或者在 OnOpenDocument()函数中调用 OnNewDocument()函数初始化空文档，并完成打开功能。

创建文档后，就需要创建视图，并完成视图初始化。读者可以通过重写 CView 类的 OnInitialUpdate 成员函数初始化视图。如果要使文档内容每次发生改变时，都能重新初始化或调整视图显示时，需要重载视图类的 OnUpdate()函数。文档的生命周期如下步骤所示。

（1）调用文档的构造函数，动态创建文档对象。

（2）每个新文档，都会调用 CDocument 类的 OnNewDocument()函数或 OnOpenDocument()函数，用户应该在其中初始化自定义数据。

（3）在文档的生命期内，用户可以与其进行交互。一般情况下，用户通过视图操作、选择和修改文档数据。视图将用户修改传递给文档，文档保存修改并更新对应的视图的显示。在这期间，文档和视图都可以接收并处理命令。

（4）当文档关闭时，框架首先调用 DeleteContents()函数，虽然重写视图的析构函数可以完成释放内存的工作，但是内存释放最好在 DeleteContents()函数中处理。

（5）调用文档类的析构函数。

在单文档应用程序中，上述第（1）步只在文档第一次创建时，执行一次。第（2）～（4）步在每次打开新文档时，都会重复执行。新文档会重复使用已经存在的文档对象。最后当程序结束时，执行第（5）步。

9.1.4　多文档应用程序框架

一般情况下，一个应用程序包含一个文档，此文档对应一个视图，并统一由一个框架窗口管理。但是有些程序需要同时支持多文档。主要分为以下几种情况。

1．多文档类型应用程序框架

应用向导为用户创建的是单文档类，但是，有时候用户需要多于一个文档类型。如应用程序同时需要工作表和图表文档两种类型的文档。每种文档类型由自己的文档类表示，并且有与其相应的视图。当用户选择 File|New 命令时，框架显示一个对话框，其中列出支持的文档类型。然后用户选择要创建的文档类型。每种文档类型由自己的文档模板对象管理。要创建额外的文档类，则可以使用类向导，新增一个派生自 CDocument 类的派生类，设置文档信息，然后实现新类的数据。要使框架识别文档类，则必须在应用程序类的 InitInstance()重载函数中调用 AddDocTemplate()函数将新文档类型注册到框架对象中。

2．多视图类型应用程序框架

一般情况下文档使用单视图，但是经常会遇到单文档支持多视图的情况。要实现多视图，文档对象需要保存一个视图列表，提供增加视图和移除视图的成员函数，并提供UpdateAllViews()函数，允许当文档数据改变时，通知多个视图。在 MFC 中存在以下 3 种类型的多视图结构。

- 在多文档框架中，支持多个同类型视图对象。此种结构下，用户选择 New|Windows 命令打开相同文档的新视图框架，然后可以通过多个视图框架查看同一文档内容。
- 在单文档框架中，支持多个同类型视图对象，即分割对话框。它将单文档对话框的视图空间分为多个视图。此种方式下，框架会从同一个视图类中创建多个视图对象。
- 在单文档框架中，支持多个不同类型视图对象。此方式下，多个视图共享单个框架对话框。视图从不同类中构造，每个视图提供对相同文档的不同方式的视图。如一个视图在普通模式下显示字处理文档，而另一个视图可以在全屏模式下显示。

9.2　开发文档/视图结构应用程序

9.1 节分析了文档/视图应用程序的结构。本节将深入分析文档/视图结构应用程序的实现细节和过程。

9.2.1　目标

使用文档/视图结构的 MFC 应用程序的最大优点就是此结构支持同一文档的多个视图机制。例如当需要在电子表格和电子图表中显示同一组数字数据时，经常会遇到此种情况，在编辑电子表格时，需要同步在图表中显示。此时，就可以通过在分割的框架对话框或单个对话框的分割面板中显示这些视图实现。

上例中电子表格视图和图表视图是基于不同视图基类 CView 的。这两种视图可以与同一种文档对象相连。文档存储数据，所有的视图访问文档，从文档中获取数据并显示数据。当用户更新其中的一个视图，则视图对象会调用 CDocument::UpdateAllViews()成员函数，此函数通知与文档相关的所有视图，每个视图会从文档中读取最新的数据，并显示出来。单独调用 UpdateAllViews()，可以同步所有不同的视图。

如果不使用文档/视图结构，而在视图中直接存储数据，则实现这种情况比较麻烦。框架也为此机制的实现提供了很好的协调。这也是文档/视图结构的主要目标。使用文档和视图还可以完成以下目标。

- 管理和显示应用程序的数据。
- 提供文档数据访问接口。
- 完成文件读写。
- 完成打印。
- 处理应用程序的命令和消息。

文档用于管理数据，在文档类成员变量中存储数据。视图使用这些变量访问数据和显

示数据。文档的默认序列化机制负责从文件中读数据和向文件中写数据。

9.2.2　创建基本程序框架

在一般的 MFC 应用程序中，文档和视图是成对出现的。数据存放在文档中，但是视图有权访问文档中的数据。视图通过 GetDocument()函数访问文档，此函数返回文档的指针。当视图要绘制数据内容时，则需要通过此函数获取数据。

Visual Studio 2010 的应用向导为用户提供了创建文档/视图结构应用程序的向导，用于创建文档类和视图类的架构。使用类向导可以映射这些类的命令和消息，并可以在 Visual Studio 2010 的源代码编辑器中编写成员函数的代码。

应用向导创建的文档类派生自 CDocument 类，视图类派生自 CView。名字为包含工程名称的默认名，在向导中可以使用类对话框修改默认名。如建立一个名为 SDISample 的工程，则创建的文档类为 CSDISampleDoc，创建的视图类为 CSDISampleView，创建的框架类为 CMainFrame，创建的应用程序类为 CSDISampleApp。对于多文档视图结构，向导还会自动创建名为 CChildFrame 的类，用于表示子框架类。视图的功能是图形化显示文档数据，并接收用户输入，对视图的操作有以下几种。

- 重写 OnInitialUpdate()函数完成视图类的特殊初始化。
- 在视图类的 OnDraw()成员函数中，编写显示文档数据的代码。
- 重写 OnUpdate()函数完成当视图需要重绘时的指定操作。
- 连接适当的 Windows 消息和用户接口对象（如菜单项）到视图类的消息处理函数中。
- 完成用户输入的处理。
- 对于多页文档，必须重载 OnPreparePrinting()初始化打印对话框，传入打印页码和其他信息。

在视图类中可以处理鼠标或键盘事件触发的 Windows 消息，也可以处理菜单、工具栏或快捷键等命令。通过处理这些消息和命令，视图可以处理用户的输入。在程序中，可以根据实际需要处理这些命令，使用剪贴板 Edit 菜单的 Cut 命令、Copy 命令和 Paste 命令等。这些处理函数需要调用一些与剪贴板相关的成员函数在剪贴板中传输选择的数据项。

9.2.3　创建文档数据

文档最主要的作用是管理应用程序数据，因此派生自 CDocument 类的文档类中，应该增加存储文档数据的变量，并重载 Serialize()序列化成员函数，以完成文档数据到磁盘文件的读写。此外，还需要根据程序需要重载 OnNewDocument()函数和 OnOpenDocument()函数，并初始化文档的数据成员，还需要重写 DeleteContents()函数销毁动态分配的数据。

通常做法是将文档数据作为成员变量在文档类中定义，并在文档类中定义设置和获取数据成员的成员函数操作文档数据或执行更复杂的操作。如在后面的 SDISample 例子中，在文档类中定义了成员变量 m_data，用于存储提示信息。而通过文档类的 GetData()成员函数获取此成员变量的值。

视图通过 GetDocument()函数获取文档指针访问文档对象，获取文档对象后，要确保将其转换成自己的文档类型，然后通过指针访问文档的公共成员函数。当需要经常的交换数据，或不希望使用文档类的非公共成员时，则可以将视图类定义为文档类的友元。以下代码是在 SDISample 例子中，定义的数据成员及其访问函数。

```
class CSDISampleDoc : public CDocument
{
... //此处代码省略
public:
    CString GetData()
    {return m_data;};
private:
    CString m_data;
... //此处代码省略
};
```

在上例中，m_data 为文档类的成员变量，GetData()函数为获取文档数据 m_data 的成员函数。这样就实现了文档数据的只读性，不允许其他对象修改文档对象中的数据。

9.2.4　绘图操作

在文档/视图结构的应用程序中，几乎所有的绘制工作都在视图的 OnDraw 成员函数中实现，此时，用户只需按照如下步骤重写视图类即可。

（1）通过调用文档类中的获取文档数据的成员函数获取数据。

（2）通过调用框架传给 OnDraw()函数的设备上下文对象的成员函数显示数据。

当文档中的数据发生变化时，视图必须重绘反映变化，此时，视图调用文档的 UpdateAllViews()成员函数通知与文档相关的所有视图。UpdateAllViews()函数调用每个视图的 OnUpdate()成员函数，OnUpdate()函数默认重绘视图的整个客户区，程序可以通过重写此函数重新绘制发生变化的数据区。MFC 为了提高绘制速度，在 CDocument 类的 UpdateAllViews()函数和 CView 类的 OnUpdate()函数中会传入文档修改部分的信息，这样程序可以只定位到需要重绘的区域，从而提高了绘制效率。OnUpdate()函数原型为：

```
void CSDISampleView::OnUpdate(
    CView* pSender,          //事件发生对象
    LPARAM lHint,            //通过此参数可以传入需要的数据
    CObject* pHint)          //通过此参数可以传入继承自 CObject 类的派生对象
```

当视图需要重绘时，操作系统会给视图发送 WM_PAINT 消息，而视图的 OnPaint()处理函数会响应此消息，通过创建 CPaintDC 类的设备上下文对象，并调用视图的 OnDraw()成员函数。一般不需要重写 OnPaint()处理函数。

设备上下文 DC 包含设备绘制属性，如显示器或打印机信息。所有的绘制调用通过设备上下文对象生成。要在屏幕上绘制，通过 OnDraw()传入 CPaintDC 对象，要在打印机上绘制，通过 OndDraw()传入 CDC 对象建立到当前打印机的连接。在视图中绘制时，首先需要获取文档指针，然后通过设备上下文绘制。以下代码显示了 OnDraw()函数的处理过程。

```
01  void CSDISampleView::OnDraw(CDC* pDC)       //在设备上下文中绘制
02  {
03      CSDISampleDoc* pDoc = GetDocument();    //获取与视图相关的文档
```

```
04      ASSERT_VALID(pDoc);                         //验证文档对象的有效性
05      CString data = pDoc->GetData();             //获取文档中的数据
06      CRect rect;                                 //定义区域对象
07       GetClientRect( &rect );                    //获取客户区域
08      //设置文本对齐方式为居中
09       pDC->SetTextAlign( TA_BASELINE | TA_CENTER );
10      //在客户区中间绘制
11       pDC->TextOut( rect.right / 2, rect.bottom / 2,
12                            data, data.GetLength());
13   }
```

在上面的代码中，首先调用视图类的 GetDocument()函数获取文档指针，程序在文档派生类中定义了 GetData()公共函数，用于获取文档数据，调用 GetClientRect()获取要绘制的客户区域范围，然后在设备上下文的 pDC 中的中间区域绘制文本，文本内容即为从文档类的 GetData()函数中返回的内容。程序运行效果如图 9-2 所示。



图 9-2　在视图类中绘制实例运行效果

除了可以在视图中绘制，还可以在视图中接收用户输入，如鼠标选择或键盘编辑事件。通常，视图会将用户的按键输入理解为输入数据或编辑数据。假如用户在管理文本的视图中输入字符串。视图可以获取文档指针，使用此指针将新数据传给文档中用于存储数据的变量中。单一文档具有多视图的应用程序，如编辑器的分割对话框，视图会将新数据传入文档，然后调用文档的 UpdateAllViews()成员函数通知文档的所有视图更新各自的显示，反映新数据，以完成视图同步。

9.2.5　文档序列化 CArchive

序列化就是从持久存储媒体（如磁盘中）读数据或向其中写数据的过程。基本思想是，对象可以记录当前状态，由成员变量标识序列化存储。以后，对象可以通过序列化存储中读取或反序列化对象状态重新创建对象。关键点就是对象本身应该负责读取和记录自身状态。因此，对于一个要序列化的类来说，必须执行基本的序列化操作。

MFC 中，通过 CArchive 归档对象实现序列化，可以满足很多应用程序的需求。如读取整个文件到内存的应用程序，允许用户更新文件，并写入更新版本到磁盘文件。但是有时使用归档对象是不能满足需要的，如数据库程序，只编辑大文件的一部分，程序只能写入文本文件，并且多个程序共享数据文件会发生访问冲突。此时，就需要使用 CFile 对象实现序列化。可以使用 CFile 类的 Open()、Read()、Write()、Close()和 Seek()成员函数打开文件、移动文件中指定点的文件指针、在指定点读取记录，并使得用户更新记录，然后重

新定位到相同的点，将记录写回文件中。框架会打开文件，用户可以使用 CArchive 类的 GetFile()成员函数获取 CFile 对象的指针。默认情况下，CDocument 类使用序列化处理 File 菜单的 Save 和 Save As 命令。其他影响数据的命令也可以由文档的成员函数处理。

MFC 在类 CObject 对象中内建了对序列化的支持，因此所有从 CObject 派生来的类都可以使用 CObject 的序列化协议。在序列化时，需要使用归档对象 CArchive，使用插入重载符（<<）和导出重载符（>>）完成写操作和读操作。

MFC 框架提供了序列化的默认实现，可以响应 File 菜单下的 Save 和 Save As 命令，保存文档到磁盘文件中，并可以响应 File 菜单下的 Open 命令从磁盘文件中装载。在 MFC 框架下，只需要做一点工作，就可以实现文档从文件中读取和向文件中写数据的功能。所做的主要工作就是在文档类中重载 Serialize()函数。应用向导为 CDocument 类生成了 Serialize()成员函数的结构，只需写入处理代码即可。

MFC 中，在要序列化的对象中和存储媒体间，使用 CArchive 类对象作为中间对象。此对象总是与 CFile 对象相连，包含序列化必须的信息，包括文件名以及请求操作是读操作还是写操作。对象可以使用 CArchive 对象完成序列化操作，而不管存储媒体是什么类型，与 cin 和 cout 对象类似。但是 CArchive 类是读写二进制格式不是格式化的文本。以下代码是将 SDISample 中的文档内容序列化。

```
01  void CSDISampleDoc::Serialize(CArchive& ar)            //文档序列化处理函数
02  {
03      if (ar.IsStoring())                                //如果准备存储数据
04      {
05          ar << m_data;
06      }
07      else
08      {
09          ar >> m_data;
10      }
11  }
```

上面的例子只是简单地将文档类的 m_data 数据成员的值序列化到归档对象。但是，复杂的序列化过程与之是相同的，只是复杂对象的序列化要序列化的内容比较多。虽然只是序列化此变量的值，但是它存储在文件中，并不是只有 m_data 的值，还有其他标志位。图 9-3 是使用序列化保存的文件的内容。

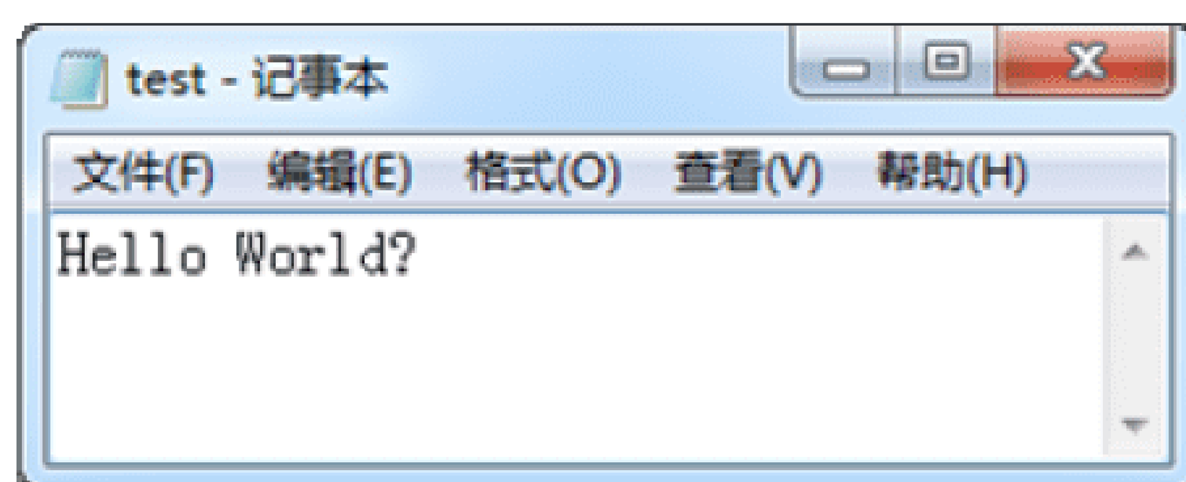


图 9-3　序列化保存的文件内容

9.2.6　让文档/视图结构支持滚动条

MFC 支持在视图中使用滚动条和自动计算显示的框架对话框的大小。当文档数据增加或用户缩小视图的框架对话框时，经常会遇到文档的大小大于视图可以显示的大小的情况，

此时需要视图必须支持滚动。任何视图都可以在 OnHScroll()成员函数和 OnVScroll()成员函数中处理滚动条消息。但是为了简化工作，MFC 提供了 CScrollView 类支持基本的滚动功能，既可以自动完成对话框和视图大小的映射，又可以自动实现滚动条滚动的功能。

读者可以自己指定当用户在单击滚动条时“滚动页”的大小，以及当单击滚动箭头时滚动行的大小。通过设置这些值可以使得滚动条功能更适合自己的程序。如在图像视图中，滚动行以 1 像素为增加单位，而在基于文本的文档中，则滚动行的大小是基于行高的。要在 MFC 中实现支持滚动条的视图，按照以下步骤操作即可。

（1）将需要支持滚动条的视图类的基类修改为 CScrollView，可以通过替换视图类的 CView 基类实现，也可以通过在向导中选择视图基类时，指定 CScrollView 类作为基类实现。

（2）在视图类中增加存储区域的 CRect 成员变量，用于记录视图显示区域。

（3）修改视图类的构造函数和 OnInitialUpdate()函数，在其中设置滚动条的参数。代码如下：

```
01  CScrollViewSampleView::CScrollViewSampleView()      //视图类的构造函数
02  {
03      m_rect = CRect(0, 0, 1024*3, -1024*3 );         //初始化视图区域
04  }
05  void CScrollViewSampleView::OnInitialUpdate()       //初始化视图类的参数
06  {
07      CScrollView::OnInitialUpdate();
08      //定义逻辑窗口大小为 40cm×30cm
09      CSize sizeTotal( 40000, 30000);
10      //定义每页的大小为 10cm×10cm
11      CSize sizePage( sizeTotal.cx/4, sizeTotal.cy/3 );
12      //定义每行的大小为 0.4cm×0.3cm
13      CSize sizeLine( sizeTotal.cx/100, sizeTotal.cy/100 );
14      //设置滚动条参数
15      SetScrollSizes( MM_HIMETRIC, sizeTotal, sizePage, sizeLine );
16  }
```

（4）在 OnKeyDown()按键处理函数中处理滚动按键的单击事件，即当用户按下 PageUp、PageDown 等按键时对应的滚动事件。如果程序不支持键盘按键的滚动事件，则此步骤可以忽略。代码如下：

```
01  void CScrollViewSampleView::OnKeyDown(UINT nChar,
02                                  UINT nRepCnt, UINT nFlags)
03  {
04      switch( nChar )
05      {
06      case VK_UP:             //向上按键
07          OnVScroll( SB_LINEUP, 0, NULL );
08          break;
09      case VK_DOWN:           //向下按键
10          OnVScroll( SB_LINEDOWN, 0, NULL );
11          break;
12      case VK_LEFT:           //向左按键
13          OnHScroll( SB_LINELEFT, 0, NULL );
14          break;
15      case VK_RIGHT:          //向右按键
16          OnHScroll( SB_LINERIGHT, 0, NULL );
```

```
17          break;
18      case VK_PRIOR:        //上页按键
19          OnVScroll( SB_PAGEUP, 0, NULL );
20          break;
21      case VK_NEXT:         //下页按键
22          OnVScroll( SB_PAGEDOWN, 0, NULL );
23          break;
24      case VK_HOME:         //开始按键
25          OnVScroll( SB_TOP, 0, NULL );
26          OnHScroll( SB_LEFT, 0, NULL );
27          break;
28      case VK_END:          //结束按键
29          OnVScroll( SB_BOTTOM, 0, NULL );
30          OnHScroll( SB_RIGHT, 0, NULL );
31          break;
32      default:
33          break;
34      }
35      CScrollView::OnKeyDown(nChar, nRepCnt, nFlags);
36  }
```

（5）在视图类的 OnDraw()函数中绘制数据内容，此处在视图上绘制一个默认样式的圆形。代码如下：

```
01  void CScrollViewSampleView::OnDraw(CDC* pDC)   //视图绘制函数
02  {
03       pDC->Ellipse( m_rect );                   //在文档中绘制圆形
04  }
```

（6）在 OnLButtonDown()函数中处理 WM_LBUTTONDOWN 消息。代码如下：

```
01  void CScrollViewSampleView::OnLButtonDown(UINT nFlags, CPoint point)
02  {
03      CClientDC dc( this );                //获取客户区上下文
04      OnPrepareDC( &dc );                  //转换上下文坐标
05      CRect rectDevice = m_rect;           //定义设备上下文区域为绘制区域
06      //将设备区域从逻辑坐标转换为设备坐标
07      dc.LPtoDP( rectDevice );
08      InvalidateRect( rectDevice );        //重绘设备区域
09      CScrollView::OnLButtonDown(nFlags, point);
10  }
```

经过上面的处理，就在程序中添加了对滚动条的支持，程序运行的效果如图 9-4 所示。

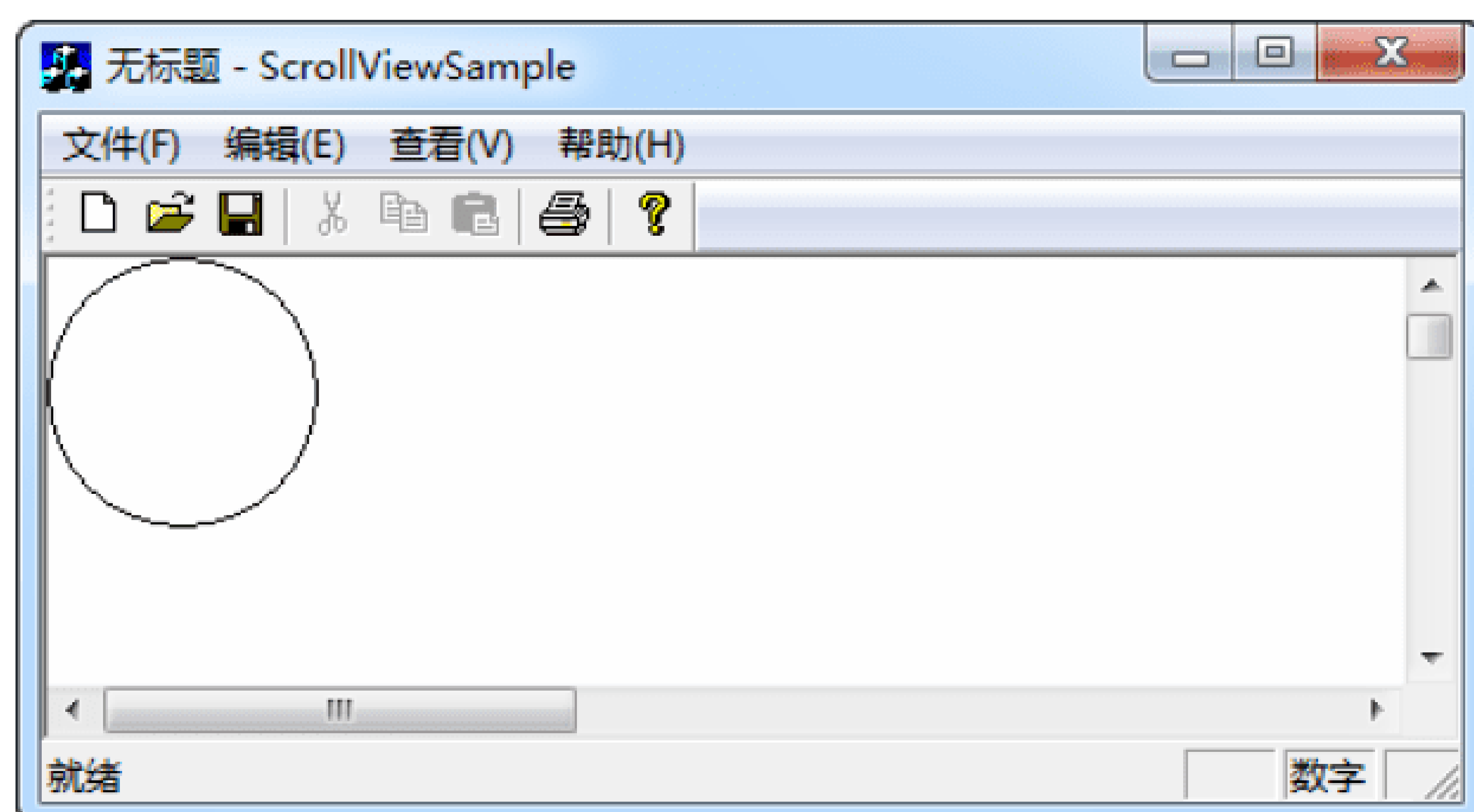


图 9-4　滚动条视图实例运行效果

9.3　对话框分割与多视图应用

通常情况下，一个框架对话框具有一个视图，但是有时一个框架对话框会包含多个视图，即分割对话框。使用分割对话框可以完成多视。本节介绍有关对话框分割和多视的基础知识，并介绍如何创建动态分割对话框和静态分割对话框。

9.3.1　对话框分割基础知识

在程序中会遇到需要分割框架窗口的情况，MFC 提供的 CSplitterWnd 类支持对框架窗口的分割，可以将框架窗口分割为两个或多个可滚动的面板。对话框分割分为两种。

- 动态分割：允许用户将当前窗口分割为多个面板，通过滚动条查看文档的不同部分，同时允许用户动态删除分割窗口。但是动态分割后的各个窗口对应的视图类的类型必须是相同的，而且采用动态分割最多将视图分割为 2×2 个子窗口。
- 静态分割：是指在程序启动时，就将窗口分割好了，每个分割后的窗口作用可以是不同的，因为静态分割允许每个面板对应的视图是不同类型的视图，支持将视图分割为 16×16 个子窗口。

9.3.2　动态分割对话框的实现

9.3.1 小节介绍了两种分割对话框，本小节将介绍如何实现动态分割对话框。步骤如下：

（1）在 MFC 应用程序向导对话框中，单击“用户界面功能”按钮，然后选中“拆分窗口”复选框，如图 9-5 所示。

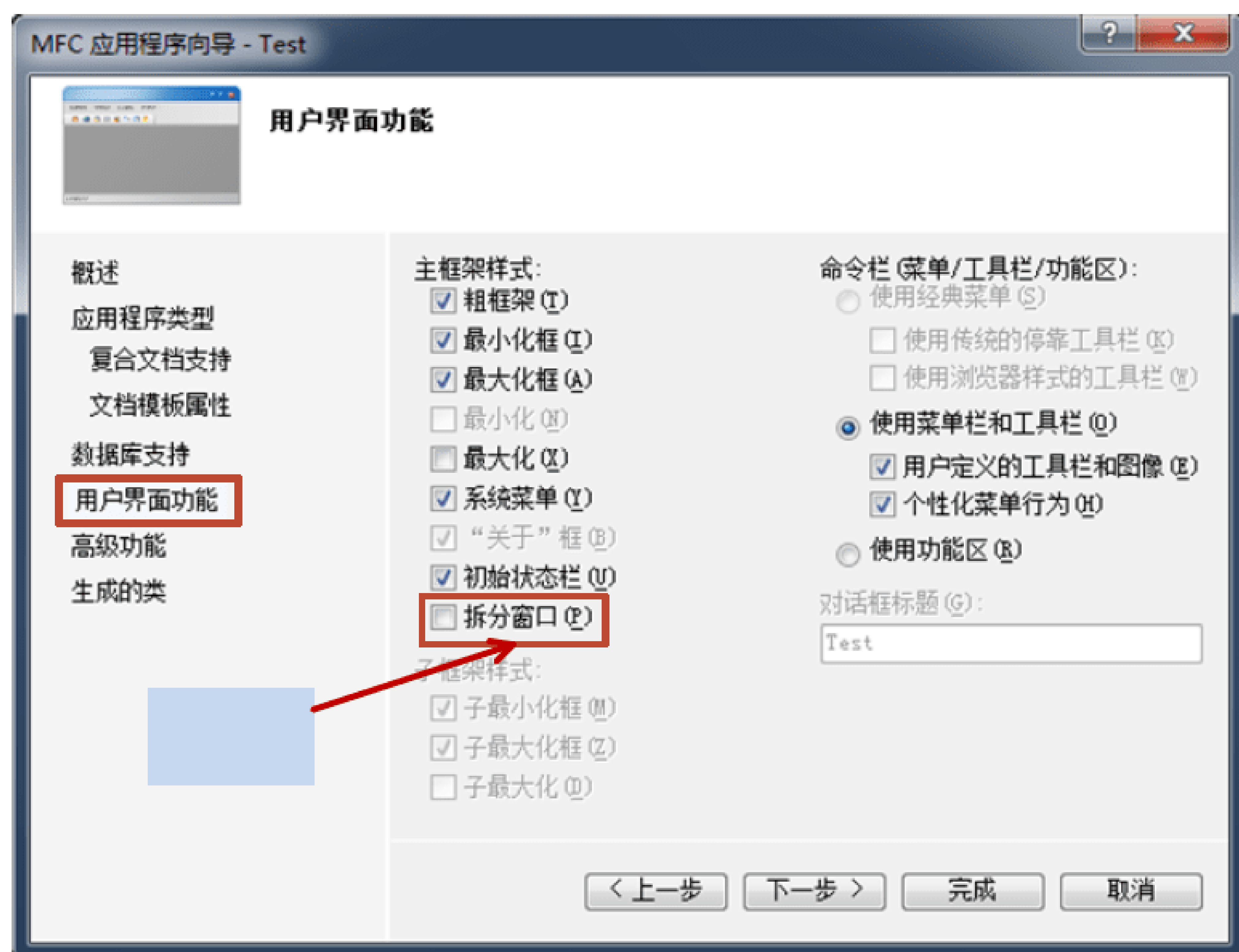


图 9-5　选择支持窗口分割的选项

（2）按照上面的步骤成功创建程序后，程序会自动添加对分割条的支持，会在框架类中定义 CSplitterWnd 类型的分割对象 m_wndSplitter，并且在框架类的 OnCreateClient()函数中会调用 Create()函数来动态分割对话框。在 Create()函数中需要指定当面板太小而不能显示全部内容时的最小行高和列宽。调用 Create()函数后，还可以通过调用 SetColumnInfo()函数和 SetRowInfo()成员函数调整最小值。代码如下：

```
01  BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
02                                    CCreateContext* pContext)
03  {
04      return m_wndSplitter.Create( this,
05          2, 2,                          //分割视图的行和列
06          CSize( 100, 100 ),             //调整最小面板的大小
07          pContext );
08  }
```

上面的代码创建 2×2 的分割窗口，运行效果如图 9-6 所示。

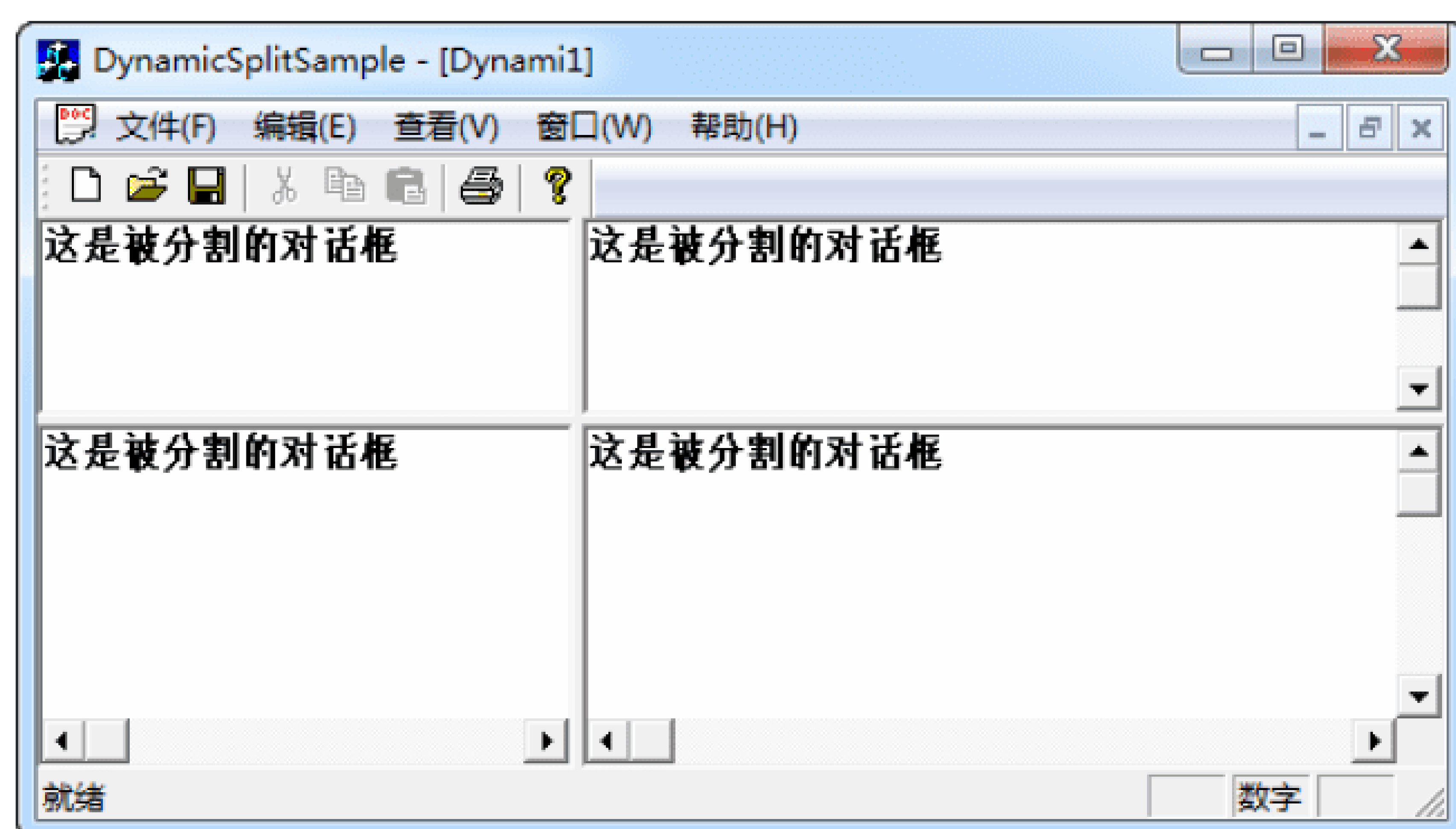


图 9-6　动态分割对话框运行效果

9.3.3　多视图的实现

静态分割对话框和动态分割对话框的思路是相同的，都是使用 CSplitterWnd 对象实现。二者不同的是，静态分割对话框最多可以支持到 16×16 规格的分割对话框，即最多可以将对话框分割为 16 行和 16 列。创建静态分割对话框的步骤如下：

（1）参考 9.3.2 小节中介绍的方法，为应用程序添加对分割对话框的支持。

（2）在程序中，根据需要添加 CSplitterWnd 成员变量，例如，在本例中，要将对话框分为 3 行，第 3 行要分为 3 列，因此定义两个 CSplitterWnd 类型的成员变量——m_wndSplitter1 和 m_wndSplitter2。

（3）重载父框架的 CFrameWnd::OnCreateClient()成员函数，在其中调用 CSplitterWnd 类的 CreateStatic()成员函数。此函数可以静态分割对话框。代码如下：

```
01  BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
02                                    CCreateContext* pContext)
03  {
04      //创建一个静态分栏窗口，分为 3 行 1 列
05      if(m_wndSplitter1.CreateStatic(this, 3, 1)==NULL)
06          return FALSE;
```

```
07      //将 CView1 连接到 0 行 0 列窗格上
08      m wndSplitter1.CreateView(0,0,RUNTIME CLASS(CView1),
09                               CSize(10,10), pContext);
10      //将 CView2 连接到 1 行 0 列窗格上
11      m wndSplitter1.CreateView(1,0,RUNTIME CLASS(CView2),
12                               CSize(10,10),pContext);
13      //将第 2 行再分为 1 行 3 列
14      if(m wndSplitter2.CreateStatic(&m wndSplitter1, 1,3,
15        WS CHILD|WS VISIBLE,m wndSplitter1.IdFromRowCol(2, 0))==NULL)
16         return FALSE;
17      //将 CView3 类连接到第 3 行的第 1 列
18      m wndSplitter2.CreateView(0,0,RUNTIME CLASS(CView3),
19                               CSize(300,200),pContext);
20      //将 CView4 类连接到第 3 行的第 2 列
21      m wndSplitter2.CreateView(0,1,RUNTIME CLASS(CView4),
22                               CSize(300,200),pContext);
23      //将 CView5 类连接到第 3 行的第 3 列
24      m wndSplitter2.CreateView(0,2,RUNTIME CLASS(CView5),
25                                CSize(300,200),pContext);
26      return TRUE;
27  }
```

上面代码在创建了分割条对象后，依次将编辑框视图、窗体视图、滚动视图、列表视图和滚动视图附加到框架中。

（4）根据需要为要附加到框架中的视图定义相关的视图类，并将声明视图类的头文件包含在框架源文件中。在相对应的视图类中执行各自的操作。编译后运行程序，运行效果如图 9-7 所示。

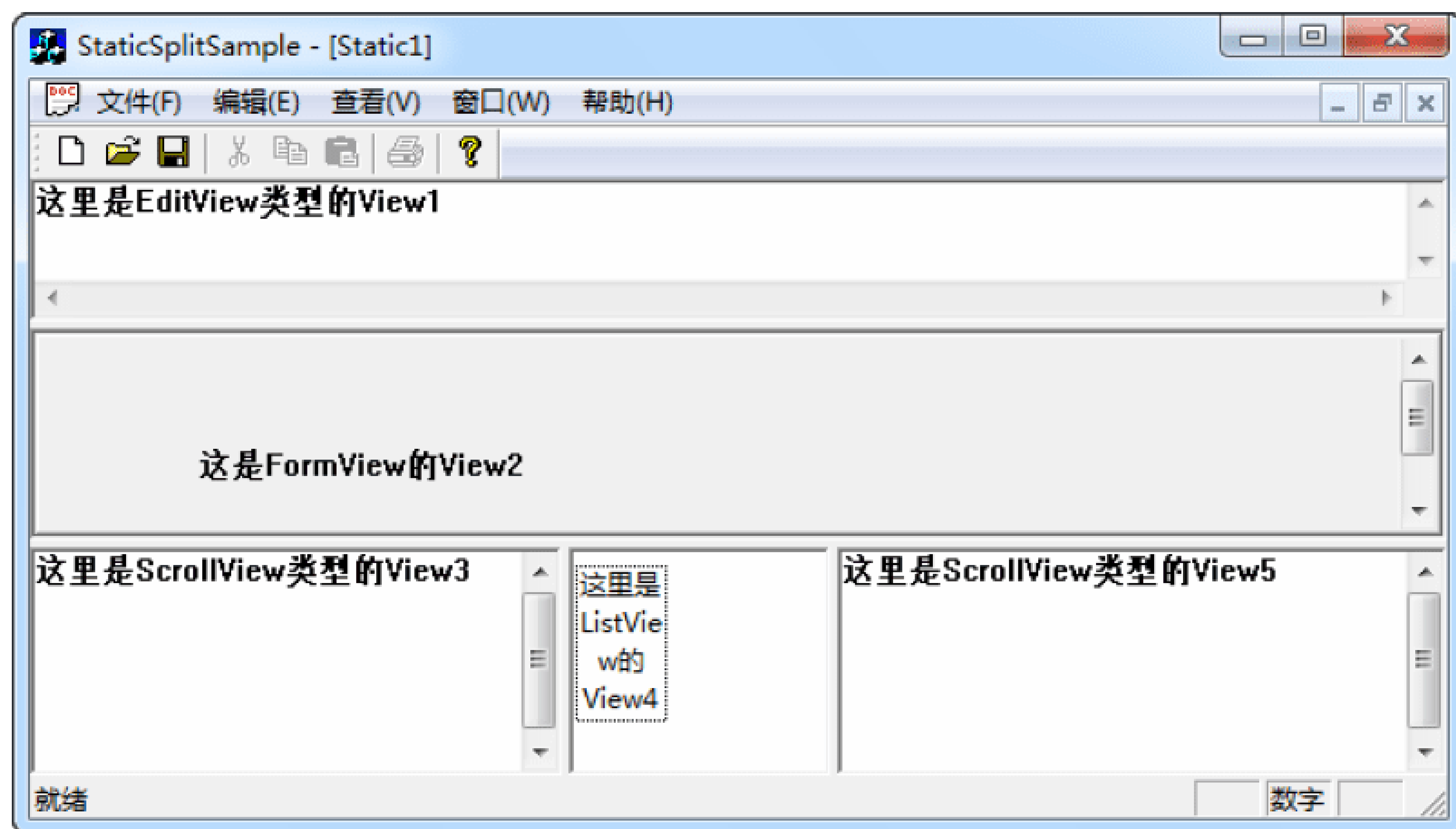


图 9-7　静态分割对话框运行效果

9.4　文档/视图应用程序实例

在本章的前面几节介绍了文档/视图结构的应用程序。本节将以一个简单的示例，讲解

如何创建文档/视图应用程序。重点叙述了如何共用文档对象的数据以及各种不同视图的菜单的定制。操作步骤如下：

（1）按照前面章节介绍的方法创建多文档/视图结构的应用程序 MDISample。

（2）在 CMDISampleDoc 类中定义需要使用的数据对象并初始化，此示例中仅以 CString 类型的 m_Data 为例，开发人员可以根据需要定义自己的数据类型对象。本例中 m_Data 的初始值为 Hello MDI！。

（3）根据需要创建自己要使用的视图类型，在视图类中添加要进行处理的代码。本示例中定义了编辑框视图 CView1 和窗体视图 CView2，并在 CView1 类中添加了以下两个函数。

```
01  CMDISampleDoc* CView1::GetDocument()          //获取相关的文档对象
02  {
03      ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMDISampleDoc)));
04      return (CMDISampleDoc*)m_pDocument;
05  }
06  void CView1::OnInitialUpdate()                //初始化视图对象
07  {
08      CEditView::OnInitialUpdate();             //调用基类的函数
09      CMDISampleDoc* pDoc = GetDocument();      //获取相关的文档类
10      //设置编辑框视图中编辑框的内容
11      this->GetEditCtrl().SetWindowText("这里是编辑框视图 CView1，文档内容="
12                                          + pDoc->m_Data);
13  }
```

同样，在窗体视图 CView2 中除了添加 GetDocument()函数外，还添加了以下函数。

```
01  void CView2::OnInitialUpdate()                //初始化视图对象
02  {
03      CFormView::OnInitialUpdate();             //调用基类的函数
04      CMDISampleDoc* pDoc = GetDocument();      //获取相关的文档类
05      //设置窗体框视图中静态框的内容
06      GetDlgItem(IDC_STATIC_1)->
07        SetWindowText("这里是窗体视图 CView2，文档内容=" + pDoc->m_Data);
08  }
```

（4）在资源中添加 CView1 和 CView2 对应的资源信息，主要是菜单和字符串 ID。本例中为 CView1 和 CView2 分别定义了菜单和字符串 ID。

（5）修改应用程序类 CMDISampleApp 的 InitInstance()实例初始化函数，在其中注册需要使用的视图类。代码如下：

```
01  BOOL CMDISampleApp::InitInstance()
02  {
03  ...
04      CMultiDocTemplate* pDocTemplate;
05      pDocTemplate = new CMultiDocTemplate(
06          IDR_MDISAMTYPE,
07          RUNTIME_CLASS(CMDISampleDoc),
08          RUNTIME_CLASS(CChildFrame),
09          RUNTIME_CLASS(CMDISampleView));
10      AddDocTemplate(pDocTemplate);
11      CMultiDocTemplate* pDocTemplate1;
12      pDocTemplate1 = new CMultiDocTemplate(
13          IDR_MDIVIEW1,
14          RUNTIME_CLASS(CMDISampleDoc),
15          RUNTIME_CLASS(CChildFrame),
16          RUNTIME_CLASS(CView1));
```

```
17        AddDocTemplate(pDocTemplate1);
18        CMultiDocTemplate* pDocTemplate2;
19        pDocTemplate2 = new CMultiDocTemplate(
20            IDR_MDIVIEW2,
21            RUNTIME_CLASS(CMDISampleDoc),
22            RUNTIME_CLASS(CChildFrame),
23            RUNTIME_CLASS(CView2));
24        AddDocTemplate(pDocTemplate2);
25    //...
26    }
```

在上面代码中，第一条 AddDocTemplate 语句将框架自带的视图 CMDISampleView 增加到框架中，第二条 AddDocTemplate 语句将自定义视图 CView1 增加到框架中，第三条 AddDocTemplate 语句将自定义视图 CView2 增加到框架中。

（6）修改代码后，重新编译并运行程序。程序运行效果如图 9-8 所示。

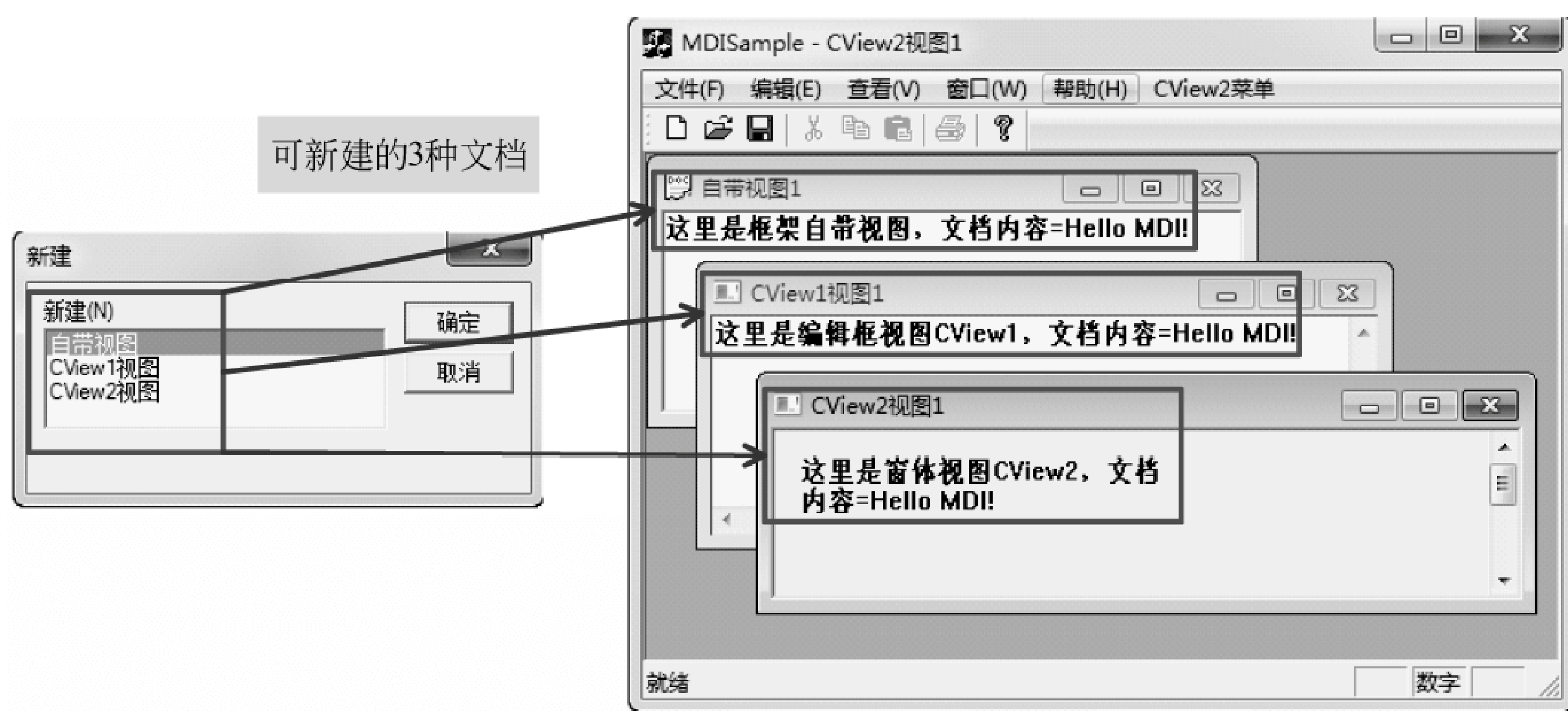


图 9-8　多文档应用程序示例运行效果

这 3 个视图除了可以共享相同的文档内容外，其他所有的特性都可以根据程序的需要进行定制。这样就完成了一个标准的多文档视图应用程序。

9.5　本 章 小 结

本章重点分析了文档/视图结构以及具有文档/视图结构的程序的开发。本章的难点是如何处理文档和视图之间的关系，设计合理的程序。同时本章还重点讲解了对话框分割和多视的实现，并以实例讲解了如何实现动态分割对话框和静态分隔对话框。最后以一个实例说明了文档/视图程序的开发过程。第 10 章将介绍另一种程序结构——对话框。

9.6　习　　题

使用 Visual Studio 2010 的程序模板生成基于单文档的应用程序，记得选中“拆分窗口”

复选框（9.3.2 小节的图 9-5），那么向导自动生成的程序会带有分割的窗口，在一行代码都没有编写的情况下编译和运行由向导生成的程序。运行效果如图 9-9 所示。完成下列操作。

（1）找到向导添加的与文档和视图有关的功能代码，并与 9.1 节和 9.2 节所学到的内容进行比较。

（2）找到向导添加的分割对话框的程序代码，并与 9.3 节所学到的内容进行比较。

（3）添加代码，实现功能：在程序左上角的视图窗口的左上角打印字符串“I’m here!”。

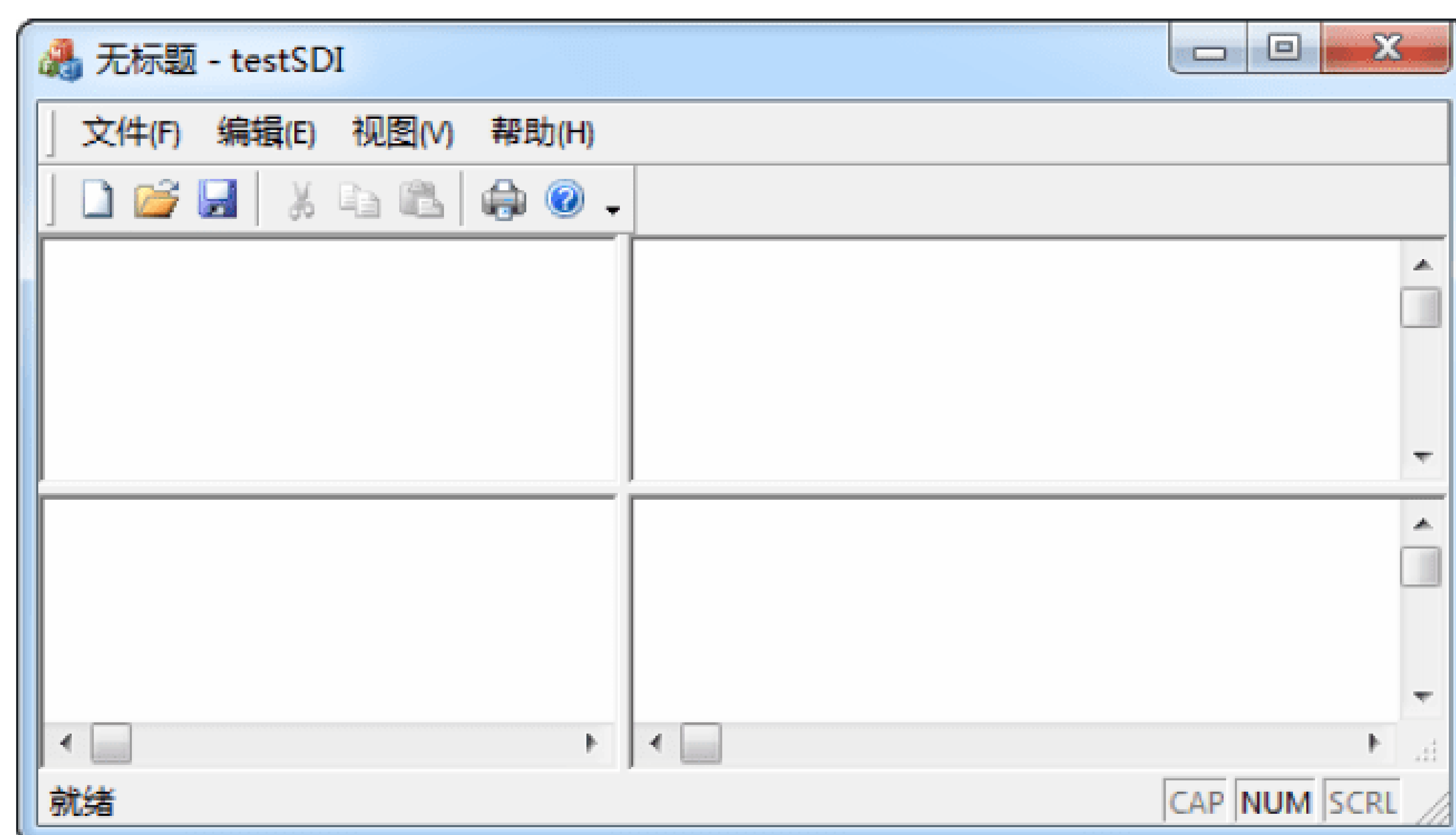


图 9-9　向导生成的对话框分割程序

【思路】参照 9.3.3 小节的示例来完成字符串的打印，需要了解打印字符串的“时机”和打印字符串的函数。

第 10 章　对话框的应用

Windows 应用程序经常通过对话框与用户进行通信，因此 VC 提供了对对话框应用程序的支持，提供了 CDialog 类管理对话框。Visual Studio 2010 对话框编辑器提供了可视化的设计对话框的方法，类向导提供了对话框中控件的初始化和验证过程，以及获取用户输入值的过程。本章将介绍有关对话框的应用。

10.1　对话框概述

在 Windows 程序中，当需要从用户处获取信息时，就需要创建对话框，如程序设置和选项。Windows 中分为两种类型的对话框：模式对话框和非模式对话框。这两者都可以包含所有类型的控件。本节就介绍有关对话框的工作方式、种类及其创建方法。

10.1.1　对话框工作方式

对话框的作用是显示信息和从用户处获取信息，即用户使用对话框与程序之间进行“对话”。当创建对话框后，发生指定事件时，程序会自动调用相应的命令处理函数，比如，接收到按键、显示信息等，用户与程序之间通过对话框不停地进行“对话”。

MFC 中通过对话框模板资源和对话框类管理对话框的实现。其中对话模板资源指定了对话框的控件和布局，指定了对话框的特性，包括大小、位置、样式和类型以及对话框控件的位置。而继承自 CDialog 的对话框类，负责在程序中管理对话框。在对话框中通过其中的控件显示、搜集信息。而在对话框中既可以包含第 7 章中介绍过的 Windows 标准控件，也可以是由第三方开发的 ActiveX 控件，还可以包含用户自己定制的控件。

虽然对话框的功能千差万别，但是创建步骤是类似的，如下所示。

（1）使用对话框编辑器设计对话框，并创建对话模板资源。在此步骤中可以根据需要添加需要包含的控件，并定制对话框和这些控件的样式、大小、位置等外观特性。

（2）使用类向导创建对话框类。在此步骤中，创建派生自 CDialog 类的自定义类完成程序的特有功能。

（3）使用类向导连接对话资源的控件到对话类的消息处理函数。在此步骤中，需要为对话框中控件添加处理函数，用于完成与用户的交互。

（4）使用类增加与对话框控件相连的数据成员，并为控件指定对话数据交换和对话数据验证。此步骤是实现用户输入与程序数据之间相连的关键步骤，只有在数据交换和数据验证中处理数据成员后，才可以将用户输入的数据真正更新到数据成员中。

在后面的小节中会详细讲述这些步骤的实现方法。

10.1.2　对话框的种类

每种对话框的功能并不相同，根据显示方法不同，MFC 将对话框分为以下 3 种类型。

- 模式对话框：此种对话框需要用户做出响应后，程序才能继续执行。用户只能在对话框打开的时候与其进行交互。对于模式对话框，处理函数在对话框关闭时，收集输入的任何数据。因为对话框对象在其关闭后，还存在，所以可以简单地使用对话框类的成员变量提取数据。
- 非模式对话框：此种对话框始终停留在屏幕上，用户任何时候都可以使用。它允许对话框打开的过程可以与其他对话框进行数据交换。用户可以在对话框打开时从其中提取数据。程序可以在任何需要的地方销毁对话框。
- 属性页：也就是标签对话框，是对话框的一种，包含拥有不同对话框控件的多个页面。每个页面的顶部有一个文件夹“标签”，单击标签会切换到标签所代表的对话框中。

10.1.3　创建与编辑对话框模板

前面介绍过对话框模板是存储对话框外观的模板资源，对于模式对话框和非模式对话框来说是一致的。在 Visual Studio 2010 中一般使用对话框编辑器创建对话框模板，步骤如下：

（1）在“资源视图”下，右击任意文件夹，在弹出的快捷菜单中选择“添加资源”命令，打开“添加资源”对话框，如图 10-1 所示。

（2）单击“新建”命令按钮，插入对话框模板资源，如图 10-2 所示。



图 10-1　“添加资源”对话框

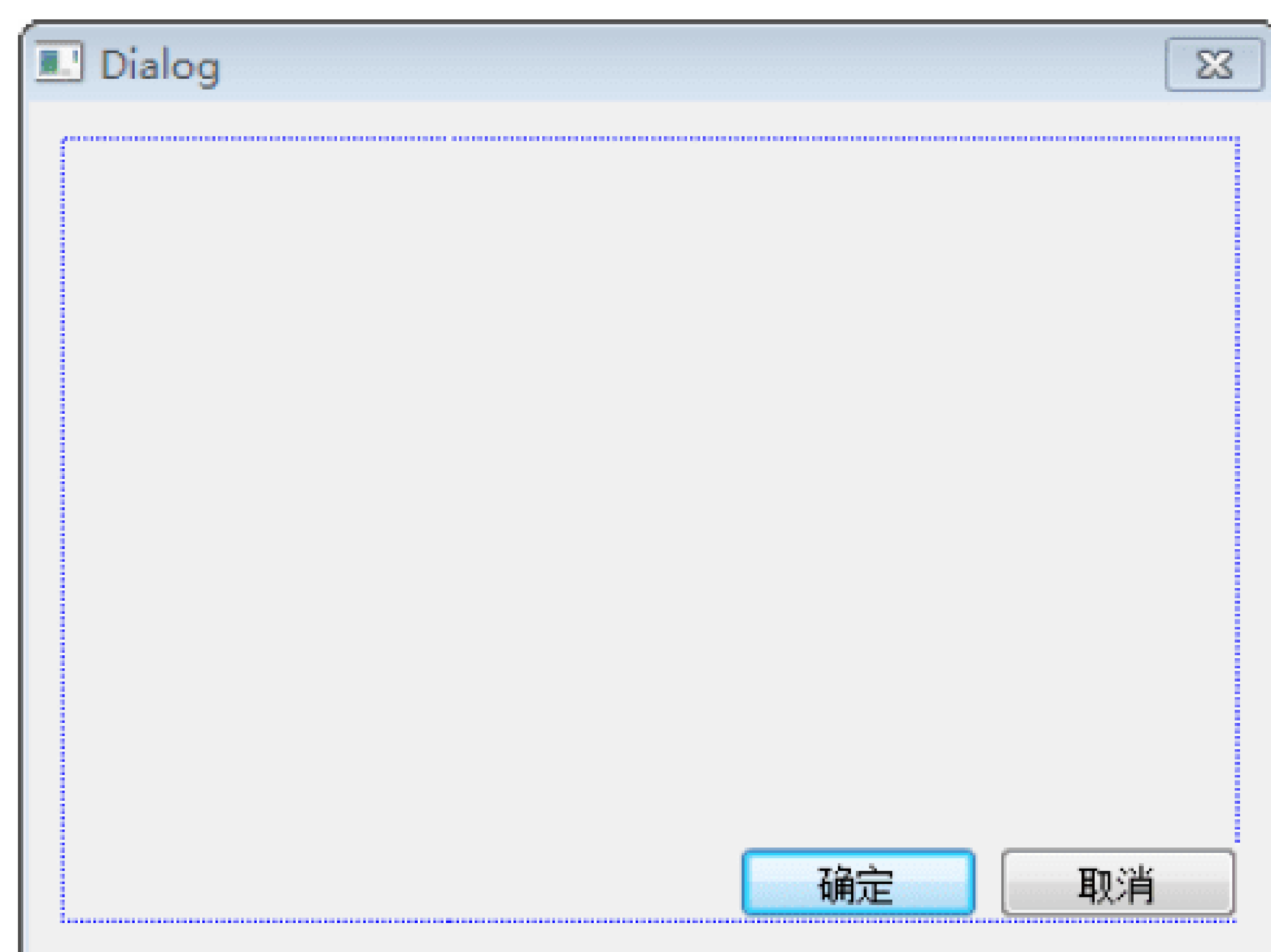


图 10-2　插入对话框模板资源

（3）在对话框模板资源中，可以拖动对话框模板资源的右下角调整对话框的大小。右击对话框模板资源，在弹出的快捷菜单中选择“属性”命令，打开“属性”对话框，如图 10-3 所示。在此对话框中可以设置对话框的位置和标题、菜单栏、对话框字体等其他与对话框相关的外观样式。

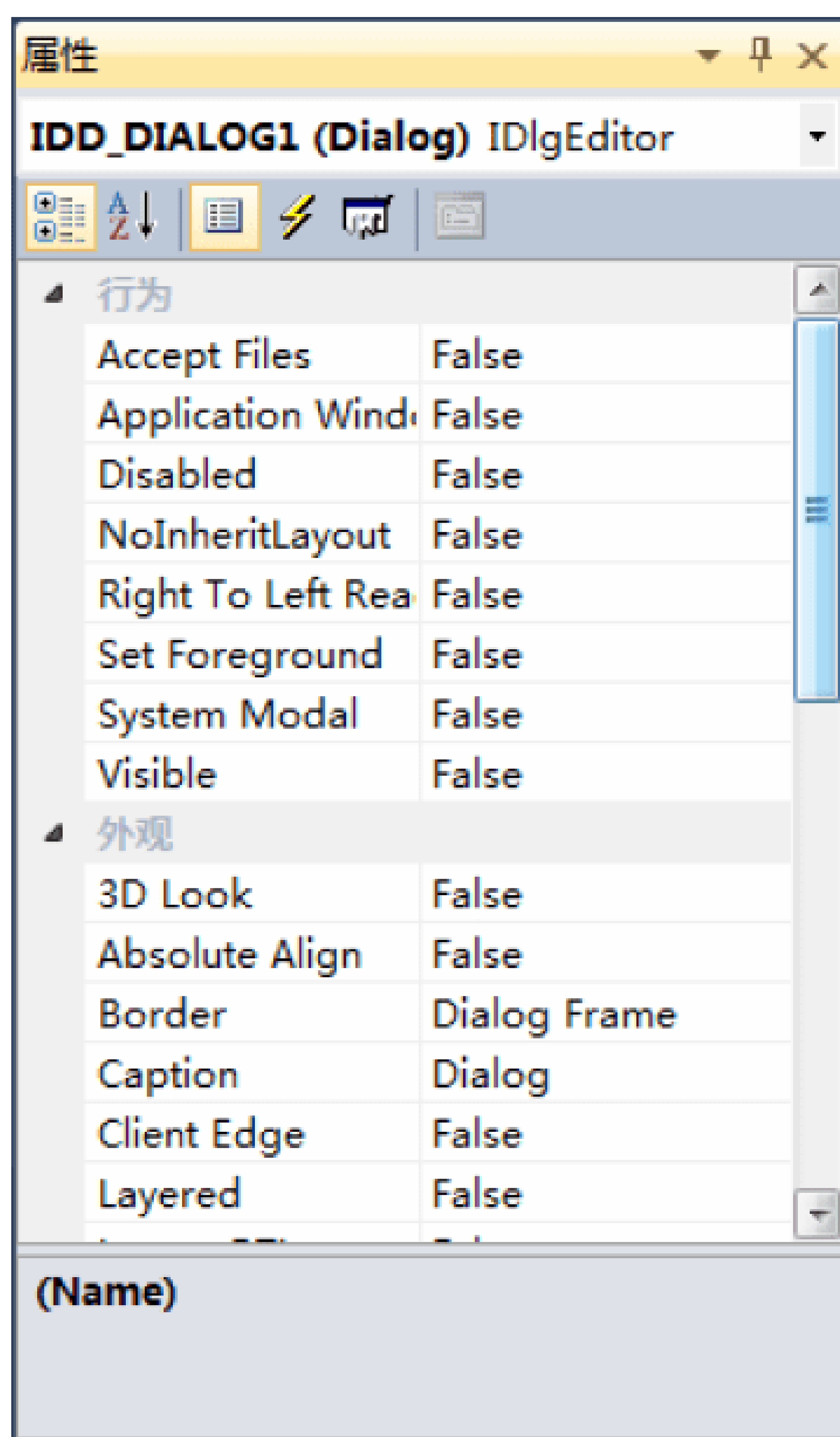


图 10-3　“属性”对话框

（4）在对话框模板资源中，可以从工具栏面板中拖动各种类型的控件，将其放置到对话框中相应的位置，并且可以使用工具栏上的调整按钮定位控件。定制完控件，可以按下 Ctrl+T 快捷键模拟对话框资源的真实使用情况。

通过上面的步骤，对话框模板资源就创建完成，Visual Studio 2010 会将其存放在应用程序的资源脚本文件中，开发人员可以在后面根据需要随时修改对话框模板资源。

10.2　对话框与程序连接

当将对话框按照要求创建后，需要使用类向导创建对应的对话框类和消息映射。要使对话框与程序相连，需要创建对话框类、映射 Windows 消息到对话框类中、为对话框添加类成员、指定对话框数据的交换以及指定对话框的数据验证等步骤。本节就依次介绍这几个步骤的实现。

10.2.1　创建对话框类

程序中的每个对话框，都需要创建一个与对话框资源一起工作的对话框类。Visual Studio 2010 为创建对话框类提供了向导。步骤如下：

（1）在 Visual Studio 2010 开发环境中，右击新添加的对话框资源，在弹出的快捷菜单中选择“添加类”命令，如图 10-4 所示。

（2）打开“MFC 添加类向导-DLGTest”对话框，如图 10-5 所示。在其中的“对话框 ID”中已经填写了对话框资源的 ID，在 Base class 下拉列表框中选择 CDialog 或 CDialogEx 类，单击“完成”按钮，添加新类。

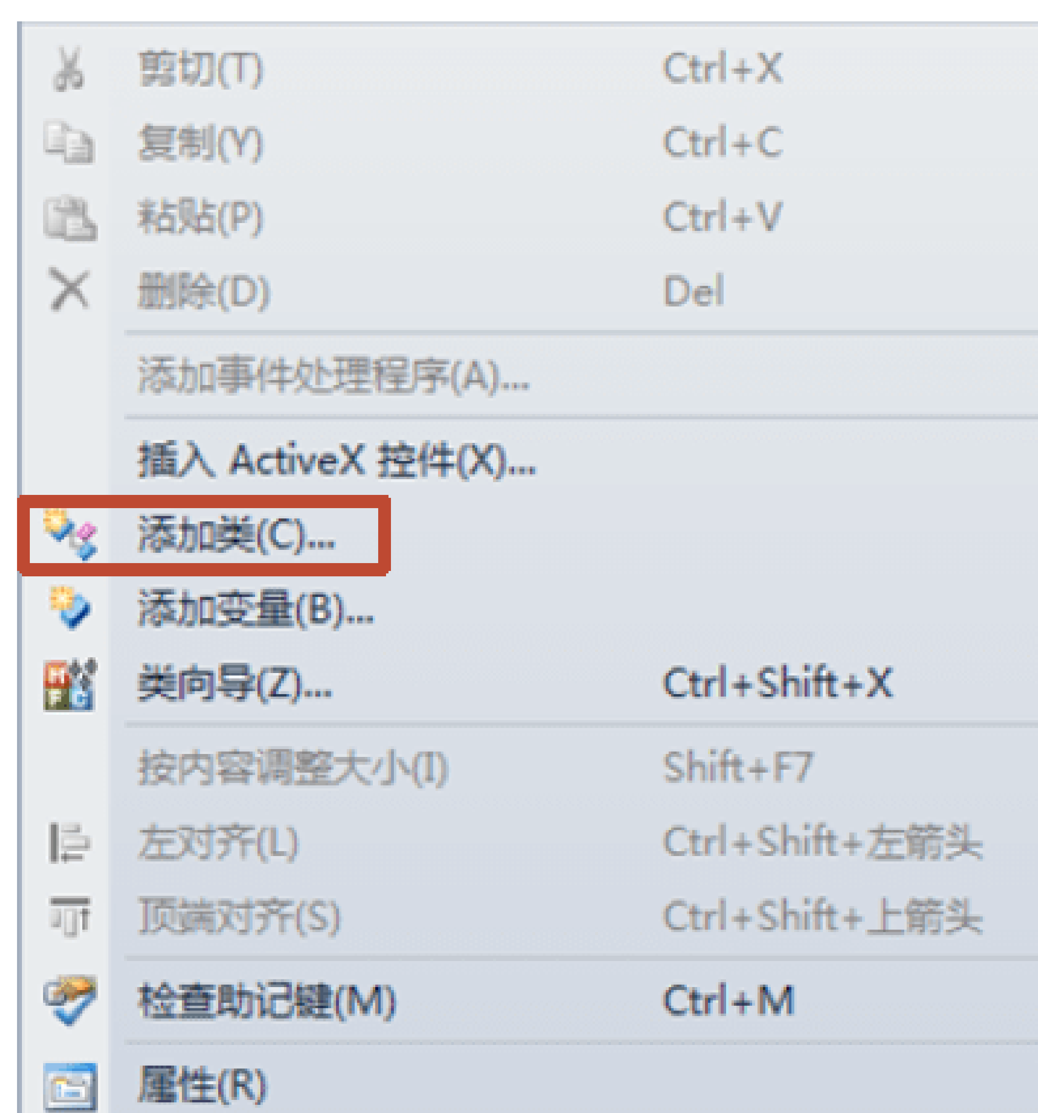


图 10-4 “添加类”命令



图 10-5 “添加类向导-DLGTest”对话框

（3）此时，向导会在工程中添加 DlgSample.h 文件和 DlgSample.cpp 文件。其中，头文件 DlgSample.h 中定义了 CDlgSample 类的声明。源文件 DlgSample.cpp 文件中包含类的消息映射、对话框的标准构造函数和 DoDataExchange()成员函数等部分。

10.2.2　为对话框类添加成员变量

创建完对话框类后，程序就可以访问控件，获取控件取值或设置控件的值。但是在获取控件时需要注意安全处理，即类型安全的访问方法。此种方法是使用内联成员函数，将类 CWnd 的 GetDlgItem()成员函数的返回类型转换成适当的 C++控件类型，代码如下：

```
01  //获取是否使用密码按钮
02  CButton* CDialogExampleDlg::GetPassCheckBox()
03  {
04      //返回是否使用密码按钮
05      return (CButton*)GetDlgItem(IDC_CHECK_PASS);
06  }
```

```
07  //设置是否设置密码选择框为选中状态
08  GetPassCheckBox ()->SetState(TRUE);
```

在上面代码中，GetPassCheckBox()函数负责将 ID 为 IDC_CHECK_PASS 的控件转换成 CButton*类型，这样在其他函数中就可以像最后一行一样安全地获取控件对象，并调用控件对应的函数。

从上面的过程中可以看出，虽然可以通过 GetDlgItem()函数安全地获取控件成员变量，但是如果在程序的多处需要获取，则代码冗余较多。因此为了简化工作量，Visual Studio 2010 提供了成员变量向导，可以完成类成员变量的添加，并自动实现安全访问。要添加的成员变量既可以是数据成员，也可以是函数成员。为对话框类添加成员变量的步骤如下：

（1）在类视图中，右击要添加成员变量的对话框类，弹出快捷菜单，如图 10-6 所示。

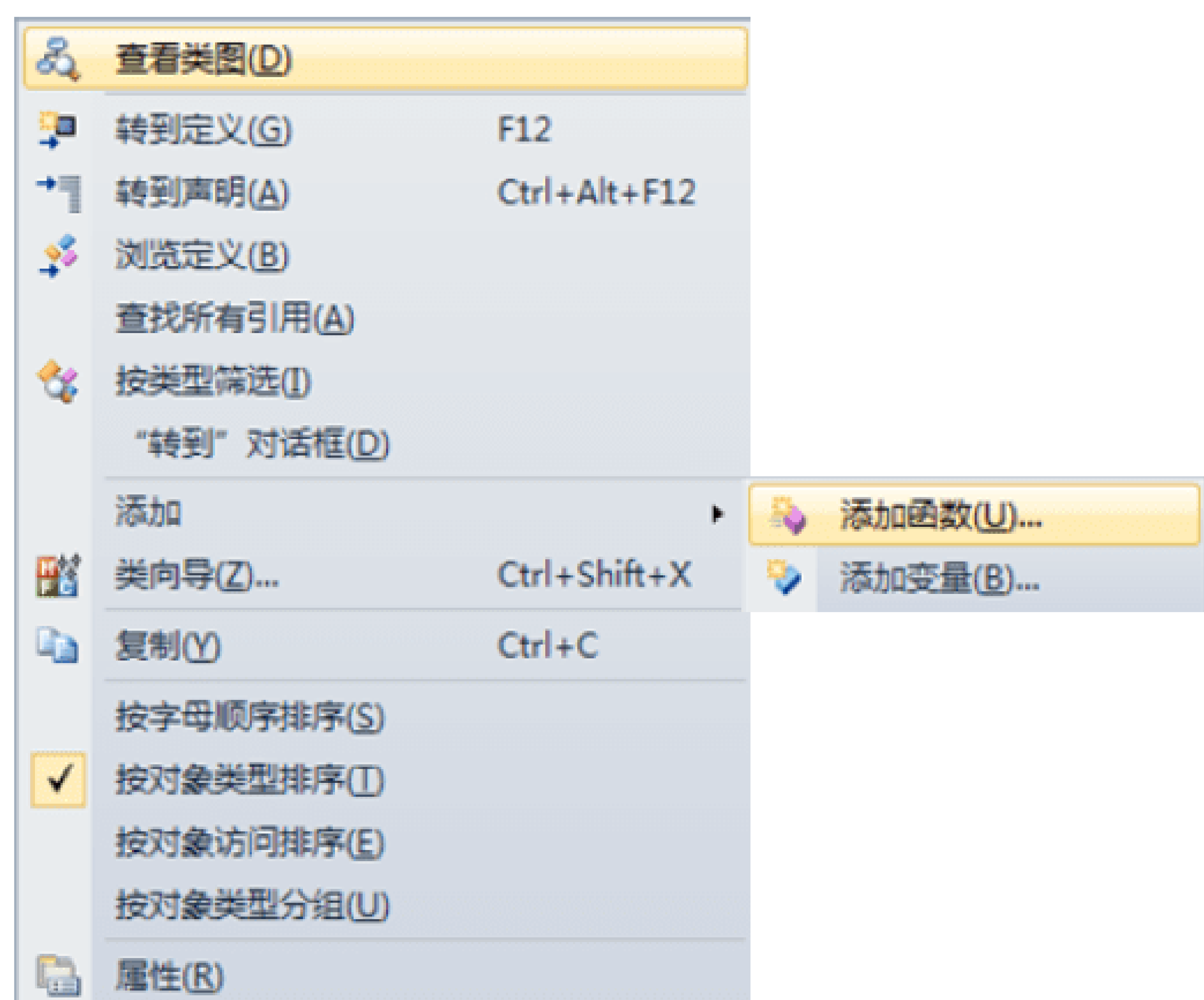


图 10-6　类快捷菜单

（2）要创建数据成员，选择“添加变量”命令，弹出如图 10-7 所示的界面。

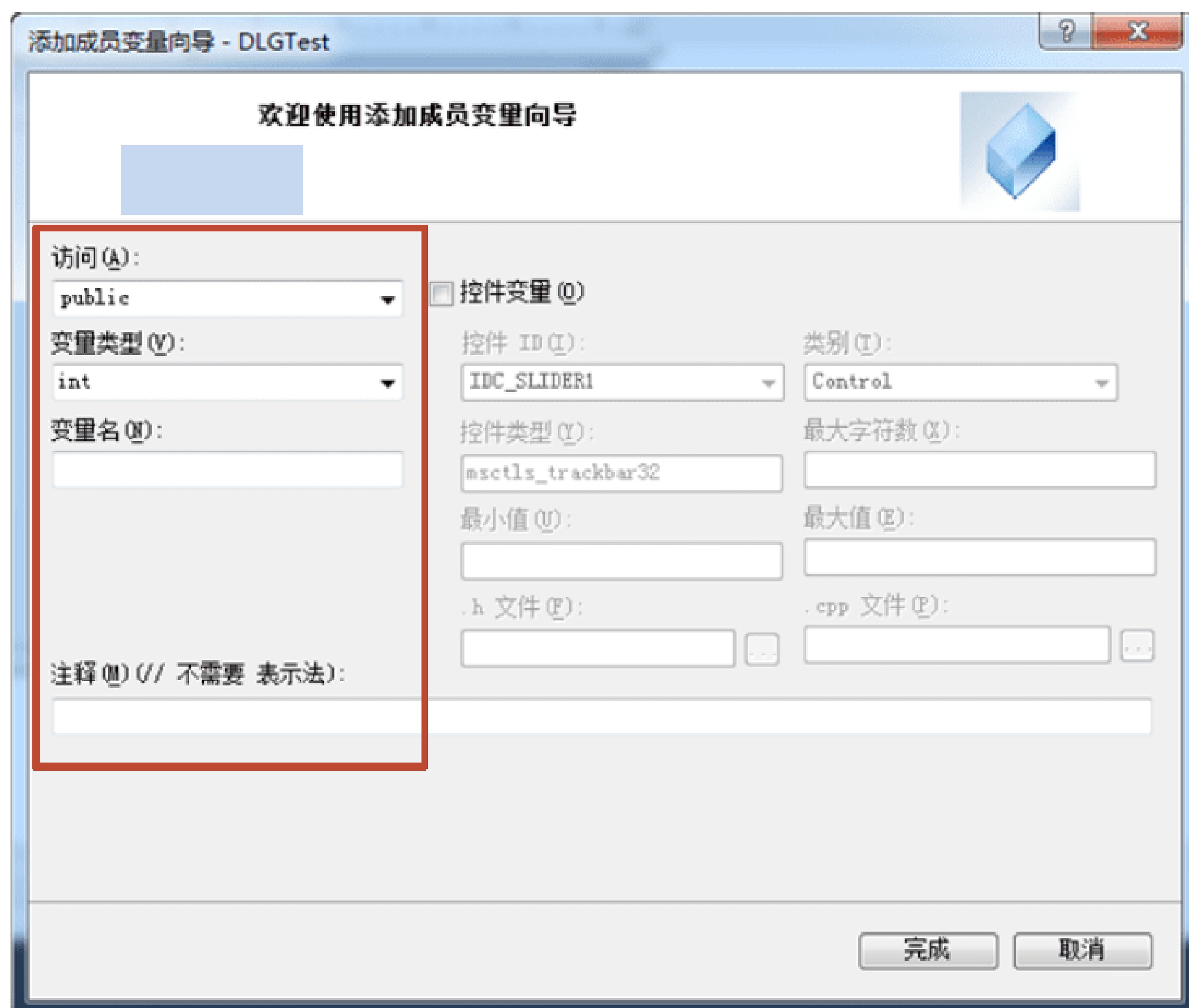


图 10-7　添加成员变量向导

在“变量类型”组合框中选择数据成员的类型，在“变量名”文本框中输入添加的数据成员的变量名，在“访问”组合框中选择要添加的数据成员的访问权限，Public、Protected 和 Private 关键字分别表示公用的、受保护的和私有的。单击“完成”按钮，完成成员变量的增加。

（3）要创建函数成员，选择“添加函数”命令，弹出如图 10-8 所示的界面。

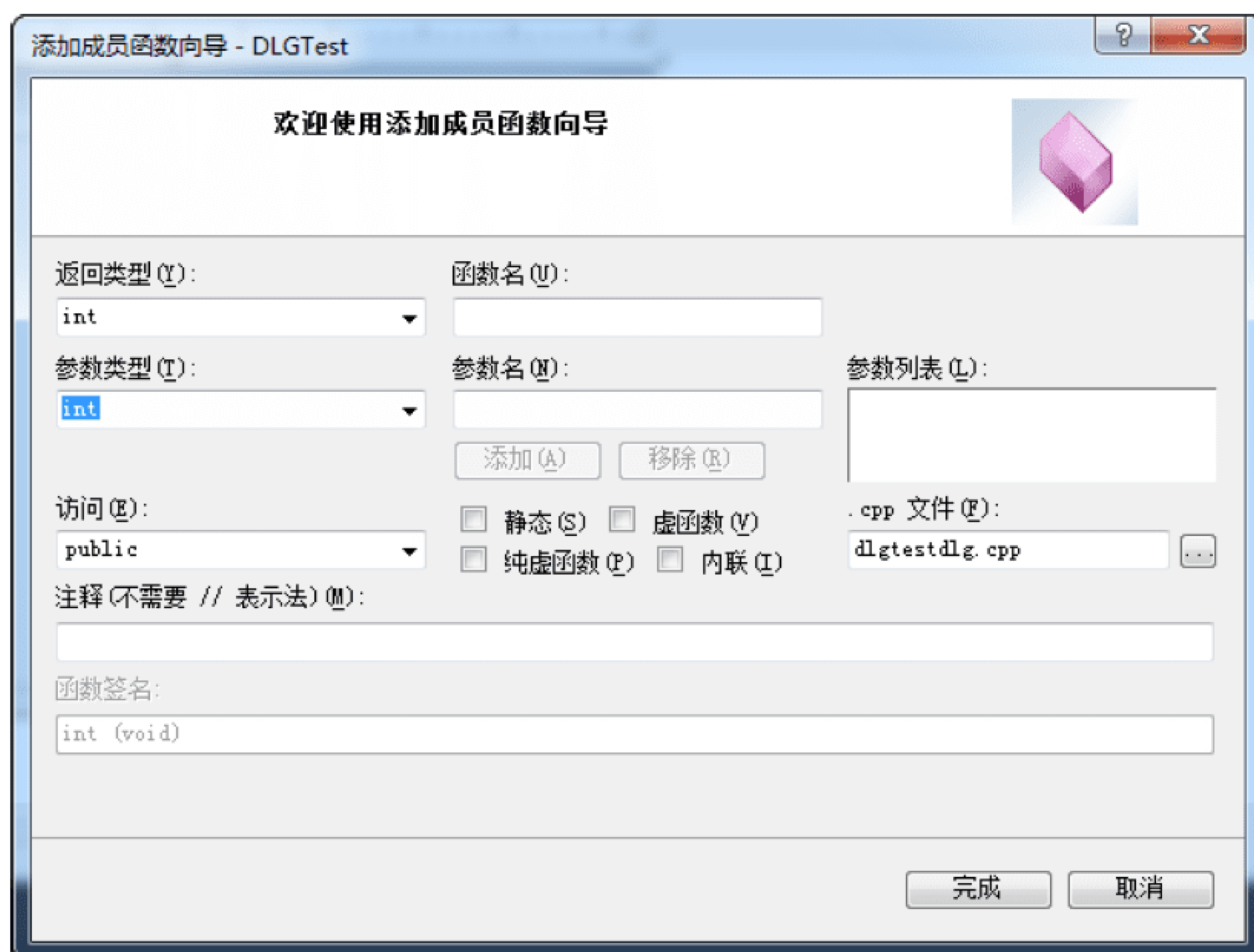


图 10-8　添加成员函数

在“返回类型”组合框中输入要添加的函数的返回值的数据类型。在“访问”组合框中选择要添加的函数成员的访问权限，Public、Protected 和 Private 关键字分别表示公用的、受保护的和私有的。在“静态”复选框中标记函数是否为静态函数。在“虚函数”复选框中标记要添加的成员函数是否为虚函数。单击“完成”按钮，完成成员函数的添加。

10.2.3　DDX 和 DDV 机制

虽然调用 CWnd 类的 SetDlgItemText()和 GetDlgItemText()成员函数或调用控件对象的 SetWindowText()成员函数和 GetWindowText()成员函数，可以设置对话框中的控件的值和获取控件当前的值，但是此方法需要手动添加，比较麻烦。因此，MFC 提供了一种简单的方法完成这两种工作，即对话框数据交换机制。

对话框数据交换（DDX）是一种初始化对话框控件中的控件和从用户处收集数据输入的简单方法，使得用户在对话框的控件与对话框对象的成员变量中交换数据更容易。要初始化对话框中的控件，用户可以设置对话框对象中的数据成员的值，框架会在对话框显示之前将值传给控件，同样可以在任何时间使用用户输入的数据更新对话框数据成员。也可以通过数据成员变量使用数据。与此同时，对话框数据验证机制自动验证对话框控件的值

的分配。

使用 DDX 机制，程序员通常在 OnInitDialog()处理函数或对话框构造函数中设置对话框对象的成员变量的初始值，对话框显示之前，框架的 DDX 机制将成员变量的值传输给对话框中的控件。对于模式对话框，如果 DoModal()函数返回 IDOK，则在对话框对象被销毁之前程序员可以获取用户输入的任何数据，如果 DoModal()函数返回 IDCANCEL，则对话框控件和数据变量中并没有进行数据交换，此时获取控件取值会产生逻辑错误。对于非模式对话框，用户可以在任何时间通过调用带有 TRUE 参数的 UpdateData()函数从对话框对象中获取数据。

对话框验证（DDV）机制是验证对话框中数据输入的有效性的简单方法。在对话框中可以使用类向导创建数据成员和它们的数据类型和指定验证规则。

MFC 为各种不同的数据类型都提供了 DDX 函数。通常情况下，在对话框类的 DoDataExchange()函数中重写 DDX 处理和 DDV 处理，代码如下：

```
01  //数据交换函数
02  void CDialogExampleDlg::DoDataExchange(CDataExchange* pDX)
03  {
04      CDialog::DoDataExchange(pDX);                  //调用基类的数据交换函数
05      DDX_Check(pDX, IDC_CHECK_PASS, m_bPass);
06      //定义是否使用密码选择框变量
07      DDX_CBString(pDX, IDC_COMBO_SEX, m_strSex);//性别组合框变量
08      //“性别”文本框中最大字符数为 20
09      DDV_MaxChars(pDX, m_strSex, 20);
10      DDX_Text(pDX, IDC_EDIT_NAME, m_strName);      //名称编辑框变量
11      //“名称”编辑框字符最大数为 50
12      DDV_MaxChars(pDX, m_strName, 50);
13  }
```

上面的例子第一句 DDX 表示 m_bPass 变量与 ID 为 IDC_CHECK_PASS 的复选框相关联，第二句 DDX 表示 m_strSex 变量与 ID 为 IDC_COMBO_SEX 的组合框相关联，第三句 DDX 表示 m_strName 变量与 ID 为 IDC_EDIT_NAME 的文本框相关联。

第一句 DDV 表示 m_strSex 变量的最大字符数不能超过 20 个，第二句 DDV 表示 m_strName 变量的最大字符数不能超过 50 个。这些 DDV 函数就是进行数据验证的函数，如果数据验证失败，则 DDV 函数会使用消息对话框提示用户，并将焦点定位到发生错误的控件中，使得用户重新输入数据。通常控件的 DDV 函数会在 DDX 函数发生后被调用。

除了使用标准的 DDX 函数和 DDV 函数，读者还可以自定义 DDX 和 DDV 函数。要注意的是，类向导会在数据映射中编写所有的 DDX 和 DDV 调用。使用 Visual Studio 2010 提供的向导，步骤如下：

（1）按下 Ctrl+Shift+X 组合键，打开“MFC 类向导”对话框，选择“成员变量”选项卡，如图 10-9 所示。

（2）在“成员变量”列表中选择要添加成员变量控件的 ID，单击“添加变量”按钮，打开“添加成员变量”对话框，如图 10-10 所示。

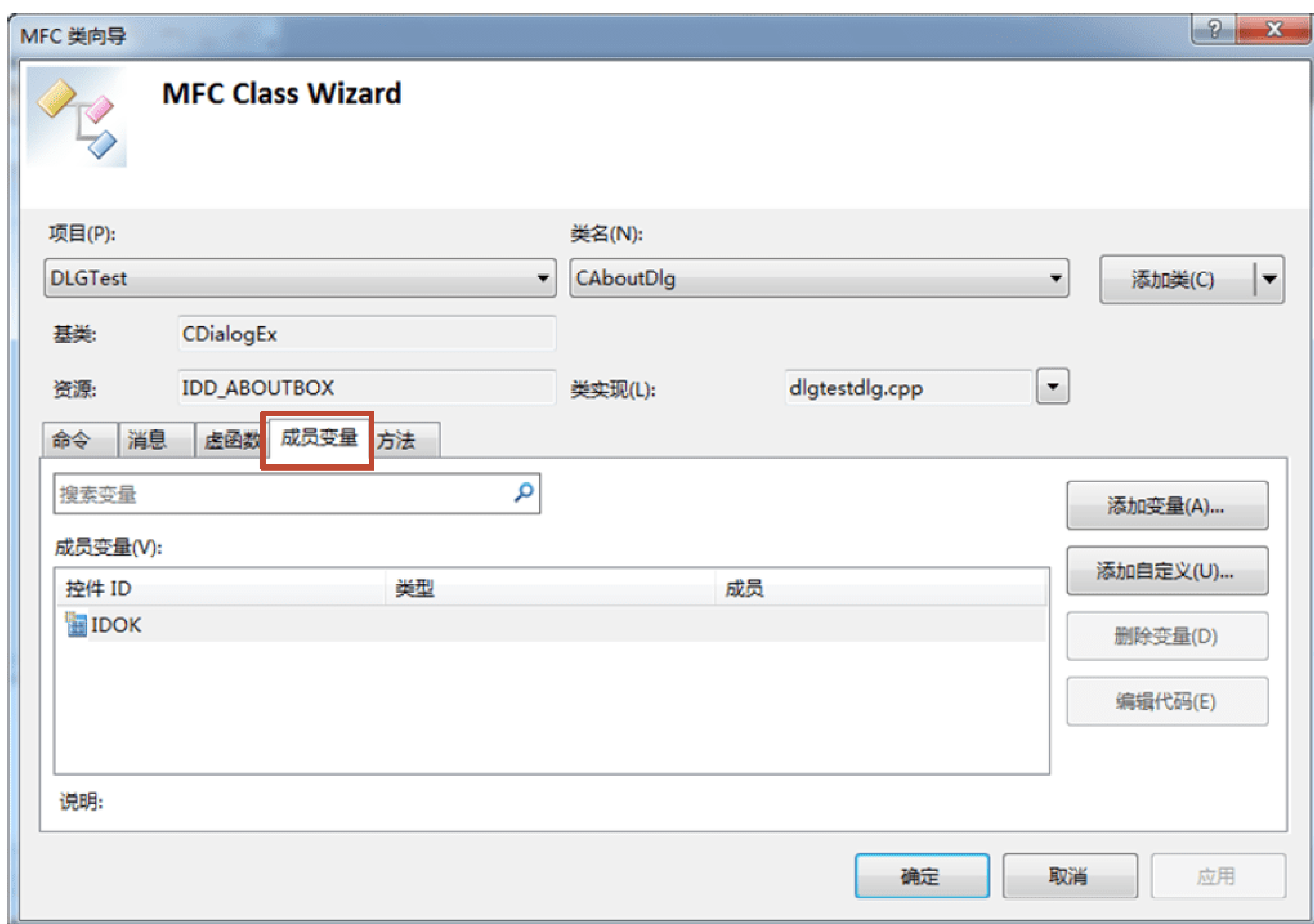


图 10-9　“MFC 类向导”对话框

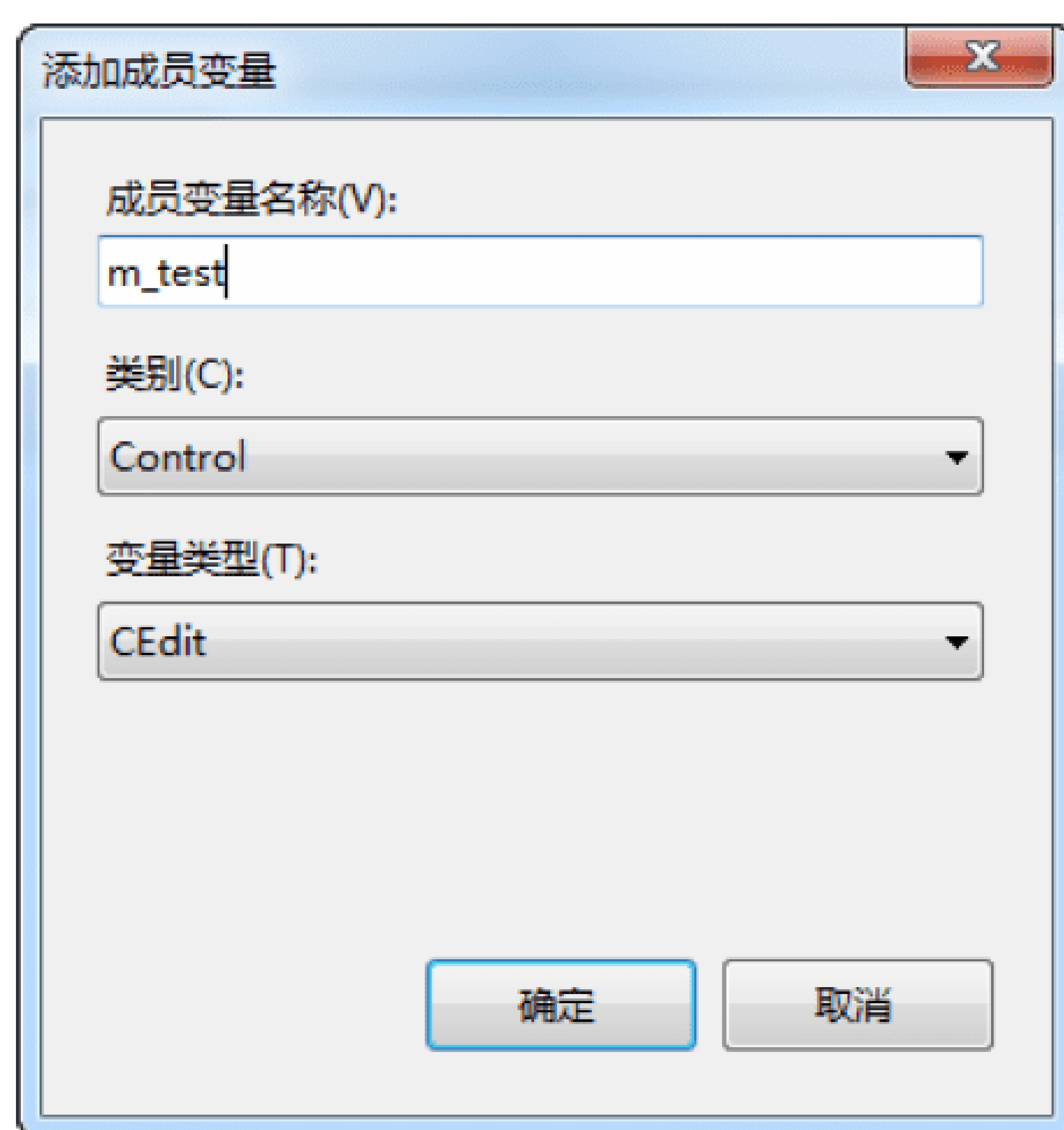


图 10-10　“添加成员变量”对话框

（3）在图 10-10 的对话框中的“成员变量名称”文本中输入成员变量的名称，在“类别”下拉列表框中选择 Value 或 Control 选项，分别代表创建的成员变量是数据值还是控件值，在“变量类型”下拉列表框中选择创建的成员变量的类型，当选中 Value 时，对话框底部出现对应的 DDV 设置。本例因为创建的是 CString 类型的数据成员，因此会出现“最大字符数”文本框，其中用于设置添加的字符串变量的最大长度，如图 10-11 所示。

（4）按照上面的（2）～（3）步骤依次添加需要的数据成员变量，单击“确定”按钮，完成数据数据交换和验证的添加。

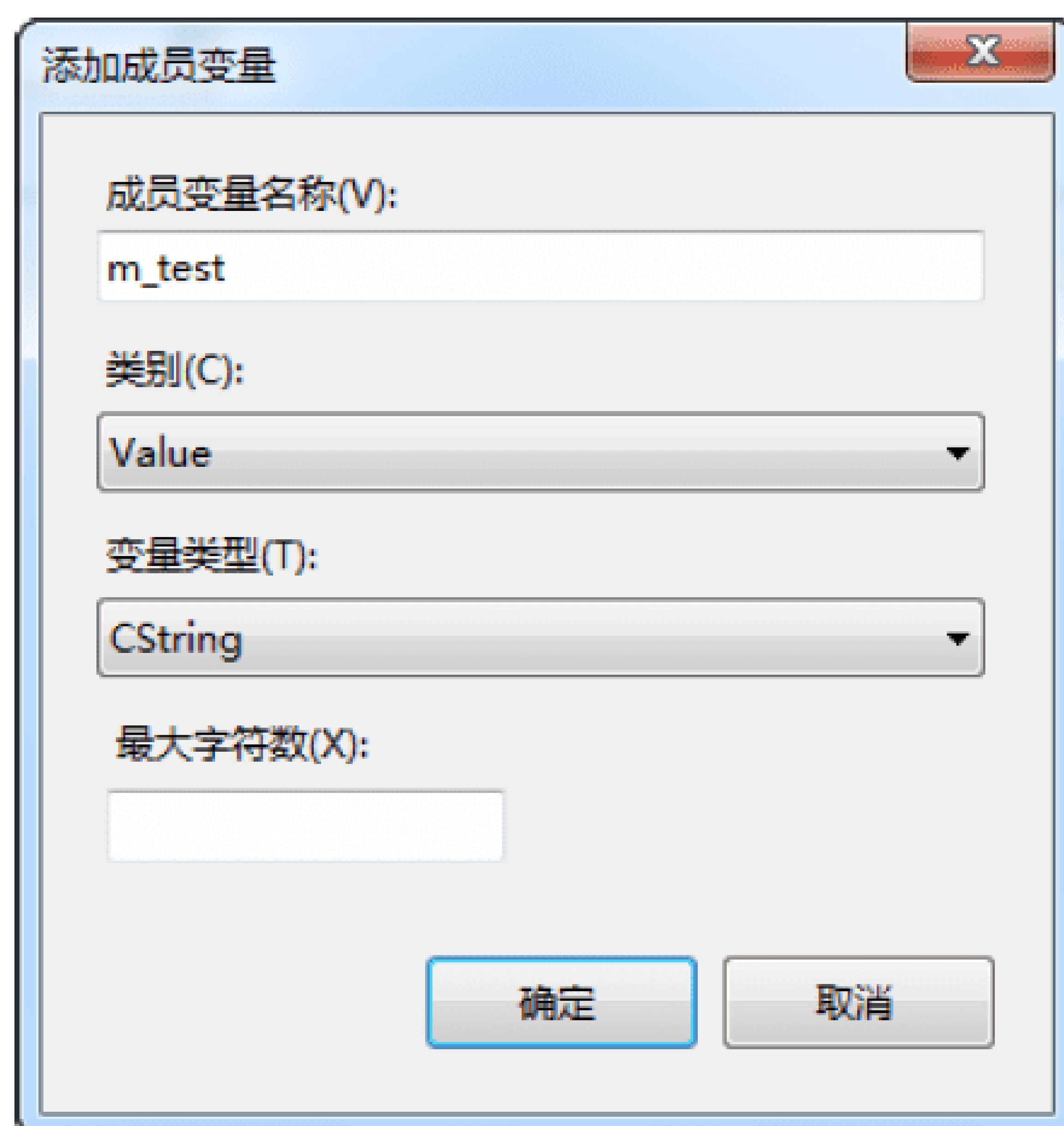


图 10-11　DDV 设置

注意：对于一个给定的控件，可以同时定义值属性成员变量和控件属性成员变量。但是只能有一个控件成员变量，因为多个对象附加到一个控件中，会导致在消息映射中模糊不清，无法定位。

10.2.4　处理对话框控件通知消息

前面几小节介绍的内容都是为对话框的使用做准备工作，对话框最主要的功能就是处理消息函数。当使用类向导创建对话框类，则向导会为类向导生成空消息映射。其中对话框可以处理的消息除 Windows 消息外，还可以处理控件消息。除手动向消息映射中增加要处理的消息外，Visual Studio 2010 提供的类向导可以映射任何想要类处理的消息或命令。为每个消息编写消息映射条目，并在类中增加消息处理函数。

在介绍控件消息处理之前，首先介绍几个 CDialog 类常用的虚函数，这些函数在通常的对话框应用中都需要对其进行重载以完成自定义功能。需要注意的是类向导不会为这些消息增加消息映射条目。

- 对应于 WM_INITDIALOG 消息的 OnInitDialog()成员函数，作用是初始化对话框控件。此函数只有在对话框显示之前才会被调用，读者必须从重载中调用默认的 OnInitDialog 处理。默认情况下，OnInitDialog 返回 true，表示焦点设置到对话框中。
- 对应于 ID 为 IDOK 的按钮的 BN_CLICKED 消息的 OnOK()成员函数，用于响应用户单击 OK 按钮的消息。这个消息函数是相对于非模式对话框而言的。
- 对应于按钮 IDCANCEL 的 BN_CLICKED 消息的 OnCancel()成员函数，用于响应用户单击 Cancel 按钮的消息。

在对话框中可以包含多种类型的控件，诸如 CListBox 和 CEdit 等。要使用这些控件，需要为这些控件对应的消息编写消息处理函数，步骤如下：

（1）按下 Ctrl+Shift+X 组合键，打开“MFC 类向导”对话框，选择“命令”选项卡，如图 10-12 所示。

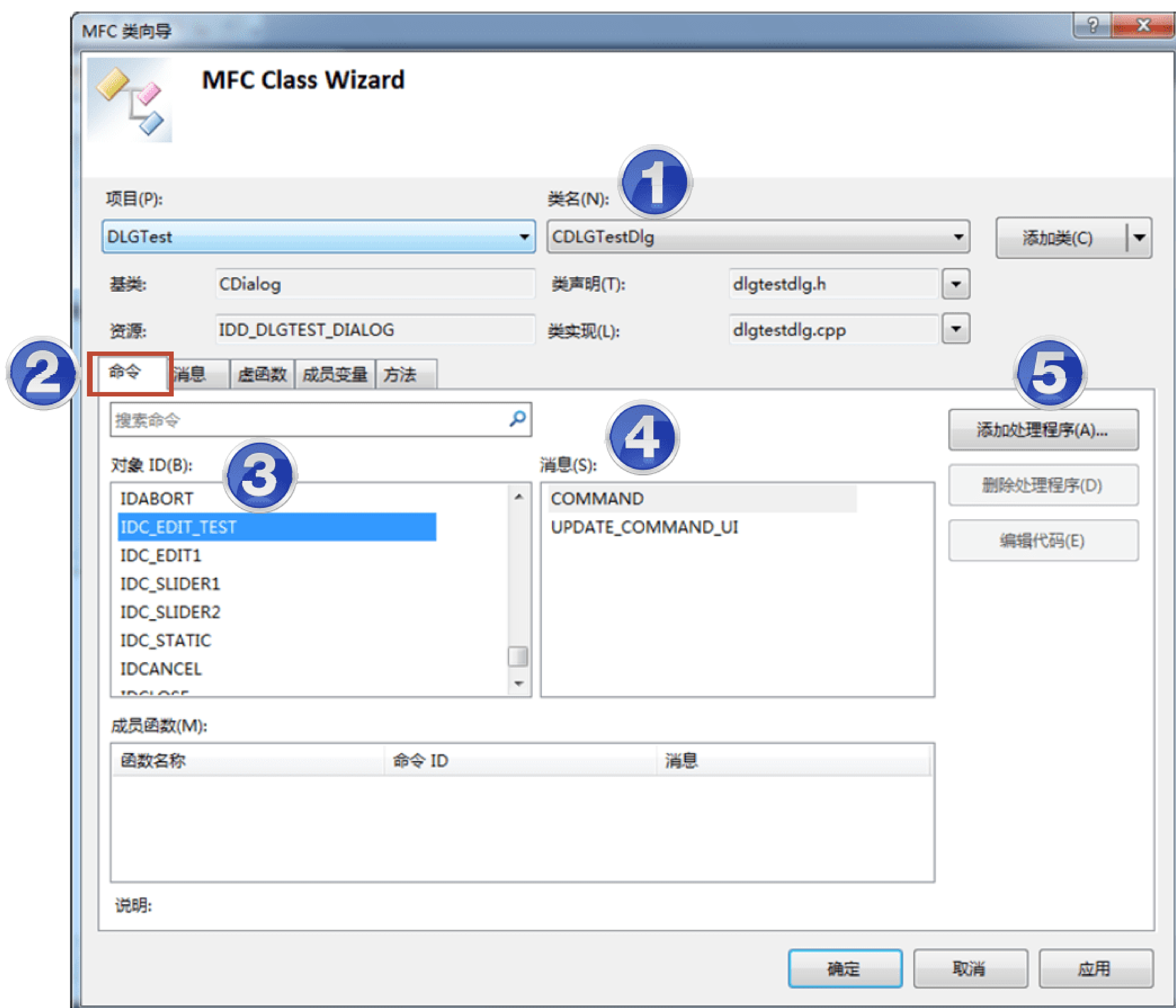


图 10-12 “MFC 类向导”对话框

（2）在对话框的“类名”下拉列表框中选择对应的对话框类，在“对象 ID”列表框中选择要添加消息处理的控件的 ID，在“消息”列表框中选择控件对应的消息，单击“添加处理程序”按钮，打开“添加成员函数”对话框，如图 10-13 所示。

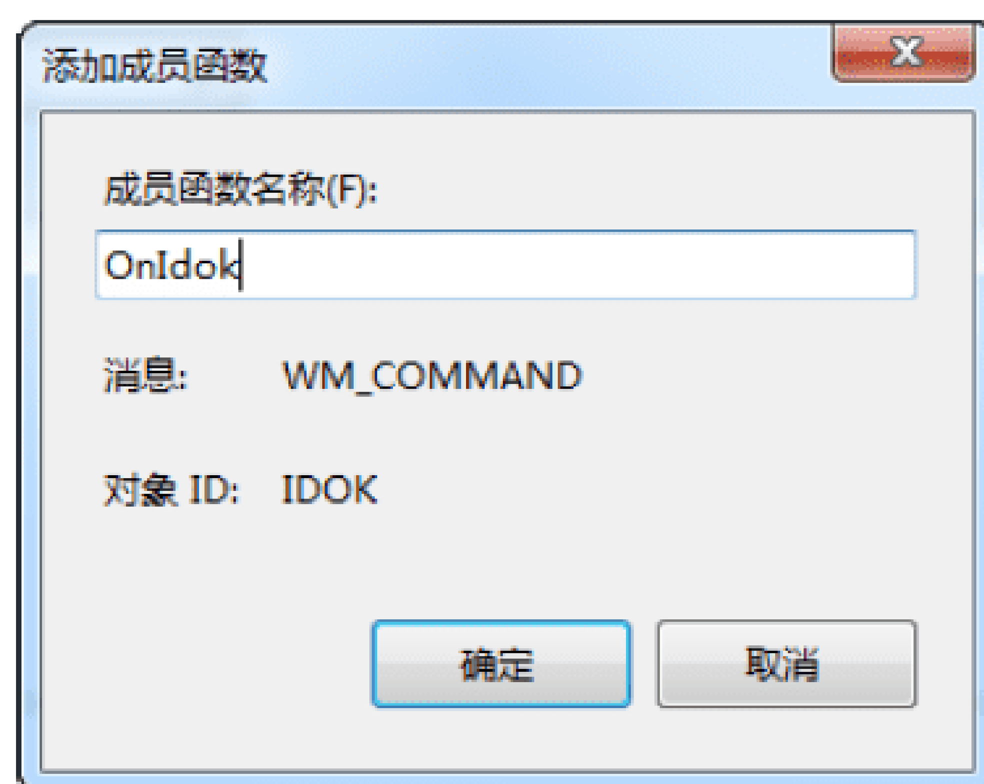


图 10-13 “添加成员函数”对话框

（3）在“成员函数名称”文本框中输入消息处理函数，单击“确定”按钮，这样就完成了控件消息处理函数的添加。

（4）在添加的函数中添加处理代码，代码如下：

```
01  void CDLGTestDlg::OnIdok()
02  {
```

```
03        //这里是文本框内容改变消息的处理函数
04    }
```

10.3　创建与显示对话框

前面介绍过，对话框可以分为模态对话框和非模态对话框，区别在于不关闭对话框的前提下，能否与其他对话框进行数据交换。需要根据不同的情况选择不同的对话框，如在绘图软件中，打开的图层管理面板使用的是非模态对话框。

10.3.1　创建模态对话框

要使用模态对话框，首先调用 CDialog 类的构造函数，然后调用 DoModal()成员函数显示对话框，再管理与用户的交互，直到用户选择 OK 按钮或 Cancel 按钮。前面介绍过需要在 OnInitDialog()函数中处理一些初始化工作。一般用户都会重写 OnInitDialog()函数，如设置编辑框的初始文本。以下代码显示了如何创建模态对话框。

```
01  //模态对话框测试按钮处理函数
02  void CDialogExampleDlg::OnButtonModal()
03  {
04      CDlgTest dlg;                          //定义测试对话框变量
05      if (dlg.DoModal() == IDOK)             //如果用户选择 OK
06      {
07          //用户单击 OK 命令
08      }
09      else                                   //否则用户选择 Cancel
10      {
11          //用户单击 Cancel 命令
12      }
13  }
```

上面代码首先定义了对话框变量，然后调用 DoModal()函数，打开模式对话框。打开的对话框就变成了程序的最顶层对话框。程序在 if 语句的两个分支中分别处理当用户单击 OK 按钮和 Cancel 按钮后需要执行的操作。示例程序运行效果如图 10-14 所示。

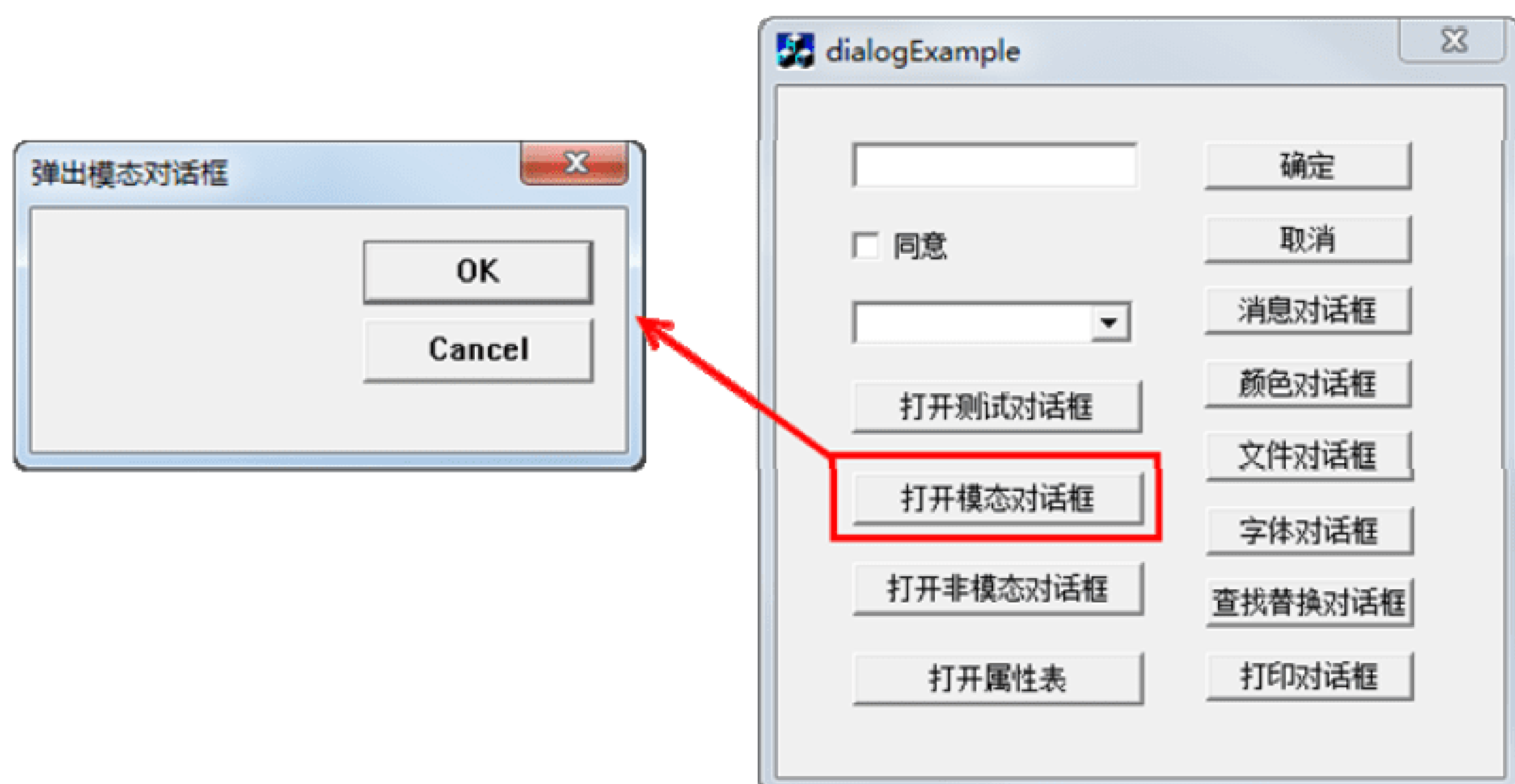


图 10-14　模态对话框

10.3.2　创建非模态对话框

要使用非模态对话框，必须在对话框类中提供公共构造函数。在创建非模态对话框时，首先调用构造函数，然后调用 Create()成员函数装载对话框资源。读者可以在调用构造函数时或调用构造函数后调用 Create()函数。如果对话框资源具有 WS_VISIBLE 属性，则对话框会立即显示；如果对话框没有此属性，则需要调用 ShowWindows()成员函数显示对话框。以下代码显示了如何创建非模态对话框。

```
01  //非模态对话框按钮处理函数
02  void CDialogExampleDlg::OnButtonNonmodal()
03  {
04      //定义测试对话框变量
05      CDlgNonModal* dlg = new CDlgNonModal();
06      //创建 IDD_DIALOG_NONMODAL 对话框
07      dlg->Create(IDD_DIALOG_NONMODAL);
08  }
```

上面代码首先定义了对话框变量，并调用 new 关键字构造对话框，然后调用 Create()函数，向其中传入对话框模板资源 ID。如果对话框没有 WS_VISIBLE 属性，则在代码的最后增加 ShowWindow()语句，程序运行效果如图 10-15 所示。

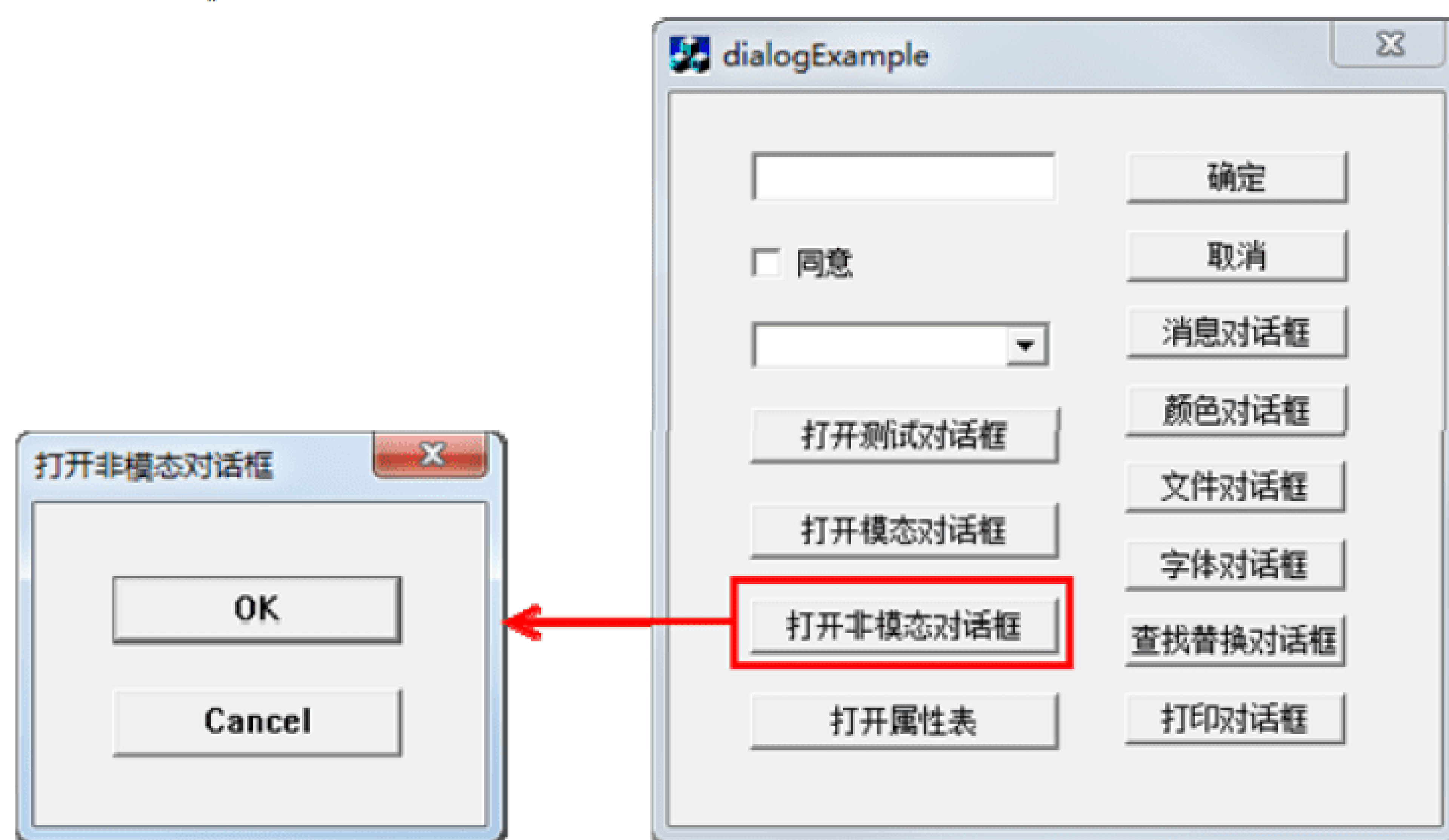


图 10-15　非模态对话框

10.3.3　修改对话框背景颜色

读者可以自己设置对话框的背景颜色，方法是在 InitInstance()重载函数中调用 CWinApp 的 SetDialogBKColor()成员函数。设置的颜色会用在所有的对话框和消息框中，代码如下：

```
01  BOOL CDialogExampleApp::InitInstance()
02  {
03     AfxEnableControlContainer();
04  ...
05      SetDialogBkColor(RGB(0,192,192));
06
```

```
07      CDialogExampleDlg dlg;
08      m_pMainWnd = &dlg;
09      int nResponse = dlg.DoModal();
10      if (nResponse == IDOK)
11      {
12          //TODO: Place code here to handle when the dialog is
13          // dismissed with OK
14      }
15      else if (nResponse == IDCANCEL)
16      {
17          //TODO: Place code here to handle when the dialog is
18          // dismissed with Cancel
19      }
20
21      return FALSE;
22  }
```

不过此种方法被 Visual Studio 2010 所“抛弃”，但它指定了另一种方式：添加对话框对消息 WM_CTLCOLOR 的处理函数，来改变对话框的颜色，代码如下：

```
01  HBRUSH CDialogExampleDlg::OnCtlColor(CDC* pDC, CWnd* pWnd,
02                                       UINT nCtlColor)
03  {
04      HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
05      //TODO:  在此更改 DC 的任何特性
06      if(nCtlColor == CTLCOLOR_DLG)
07      {
08          CBrush *brush = new CBrush(RGB(0,192,192));
09          return (HBRUSH)(brush->m_hObject);
10      }
11      //TODO:  如果默认的不是所需画笔，则返回另一个画笔
12      return hbr;
13  }
```

这样，就会将窗体的背景颜色设置为浅绿色。程序运行效果如图 10-16 所示。

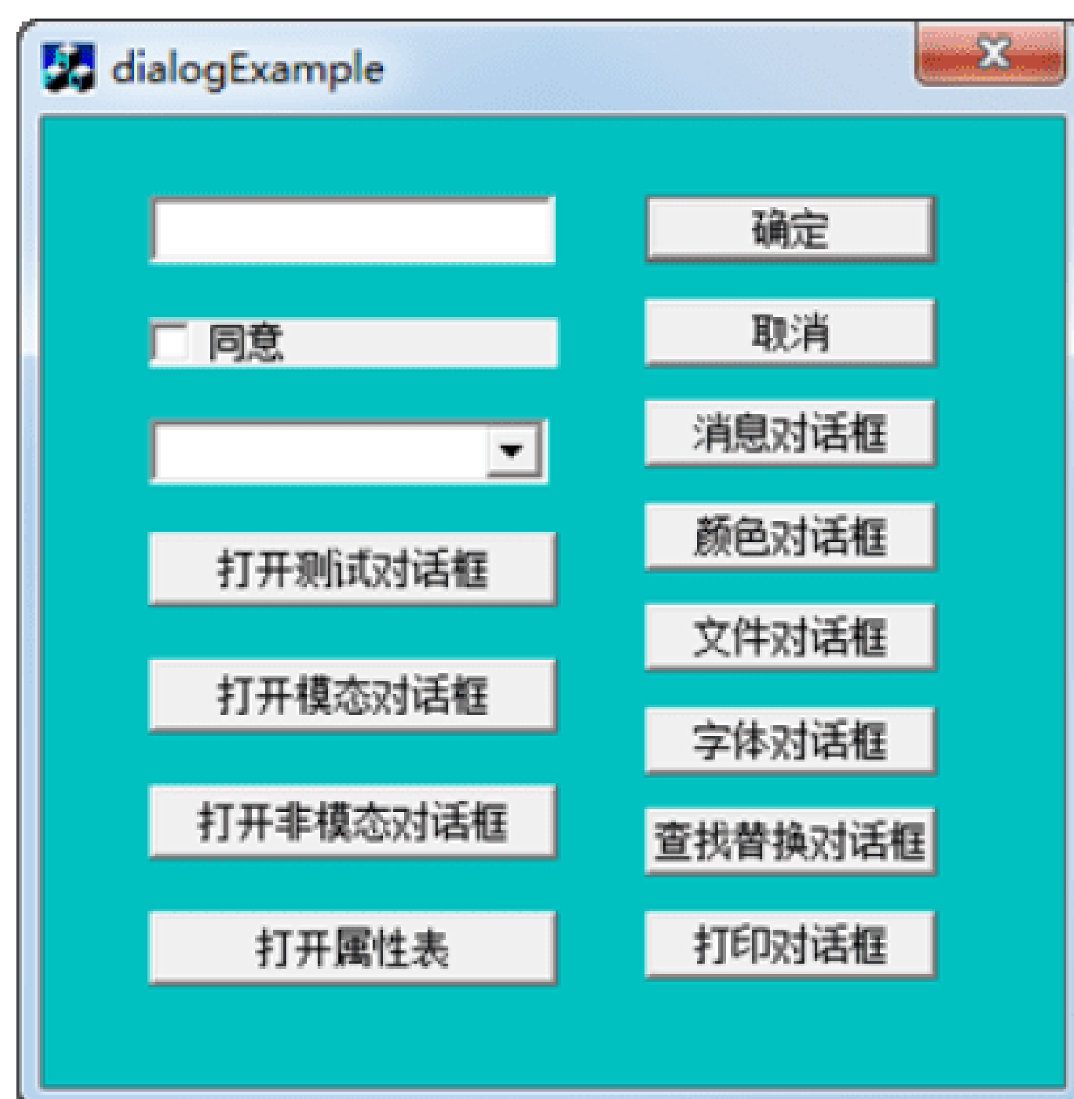


图 10-16　设置窗体背景颜色效果

10.3.4　关闭对话框

对于模态对话框，当用户单击 OK 按钮或 Cancel 按钮后，模态对话框会关闭。此时对

话框对象会发送BN_CLICKED控件通知消息，CDialog类为这些消息提供默认的处理函数：OnOK()和 OnCancel()，在其中调用 EndDialog()成员函数关闭对话框。当然，读者也可以在自己的代码中直接调用 EndDialog()函数。

对于非模态对话框，对话框的关闭动作，通常由父对话框处理。在默认的 OnClose()函数中调用销毁对话框的 DestroyWindow()成员函数。如果非模态对话框是独立的，应该重写 PostNcDestroy()函数销毁对话框对象，或重载 OnCancel()函数，从其中调用 DestroyWindow()成员函数。否则，非模态对话框的父窗口会在不需要时，自动销毁非模态对话框。

10.4　属性表对话框

属性表是一种特殊的对话框，用于完成信息分组的功能。如设置对象的属性时，可以将要设置的属性进行分组，方便使用者快速定义，提高界面友好程度。本节将介绍有关属性表对话框的使用，主要包括属性表对话框的运行机制和创建。

10.4.1　属性表对话框的运行机制

MFC 对话框可以带有属性页，也就是标签对话框。它是对话框的一种，由属性表和多张属性页组成。MFC 中的属性表对话框类似于 Microsoft Word、VC 中的对话框，看上去像包含一组带标签的表，更像一组文件夹从前到后，或一组级联对话框。前面标签中的控件是可见的。在后面标签上只有标签可见。属性表特别适合用于管理大量的属性或设置，将其清楚地分到几个分组中。通常，一个属性表可以通过替换几个独立的对话框简化用户接口。

属性表的每个页面包含自己所属的控件，基于对话框模板资源，并出现在一个“标签”上，“标签”放置在页面的顶部，用于命名页面，并指示其目的。单击属性页的标签，会将此页带到对话框的前面，将其中的控件显示出来。在 Visual Studio 2010 开发环境中，就有许多属性表的例子，比如“工具”|“自定义”对话框。在 MFC 中，属性表由 CPropertySheet 类实现，在属性表对话框中又包含多个页面，每个页面由 CPropertyPage 类实现。

10.4.2　属性表对话框的创建

因为属性表是对话框的一种，所以属性表对话框的创建与普通对话框是相同的。但是因为是特殊的对话框，因此步骤略有不同。创建属性表对话框的步骤如下：

（1）为每个属性页创建对应的对话框资源。这些属性页对话框资源的大小可以不同，框架会使用其中最大的对话框资源的大小分配属性表中每个属性页的大小。在为属性页创建对话框模板资源时，必须指定如下属性。

- ❑ 在 Caption 列表项中设置要显示在标签上的文本。
- ❑ 在 Style 列表项中选择 Child。
- ❑ 在 Border 列表项中选择 Thin。
- ❑ 设置 Titlebar 列表项为 True。

（2）按照前面介绍过的方法，使用类向导为每个属性页对话框模板创建派生自 CPropertyPage 的类。

（3）使用类向导，创建存放属性页对象的成员变量。

（4）在源代码中构造 CPropertySheet 对象，调用 CPropertySheet::AddPage()成员函数将每个页面添加到属性表中。调用 CPropertySheet::DoModal()函数或 Create()函数分别以模态方式和非模态方式显示属性表。以下是打开属性表的代码。

```
01  void CDialogExampleDlg::OnButtonOpenSheet()//打开属性表按钮处理函数
02  {
03      CPropertySheet Sheet;                     //定义属性表
04      Sheet.SetTitle("信息管理");               //设置属性表的标题
05      CPageStudent page1;                       //定义学生页变量
06      CPageTeacher page2;                       //定义教师页变量
07      Sheet.AddPage(&page1);                    //向属性表中增加学生页
08      Sheet.AddPage(&page2);                    //向属性表中增加教师页
09      Sheet.DoModal();                          //显示属性表
10  }
```

上面代码中的 CPageStudent 类和 CPageTeacher 类都是派生自 CPropertyPage 类的属性页类，并在程序中定义了对应的对话框资源。创建了这两个对象后，调用 CPropertySheet 对象的 AddPage()成员函数将这两页增加到属性表中，最后调用 DoModal()函数显示属性表，程序运行效果如图 10-17 所示。

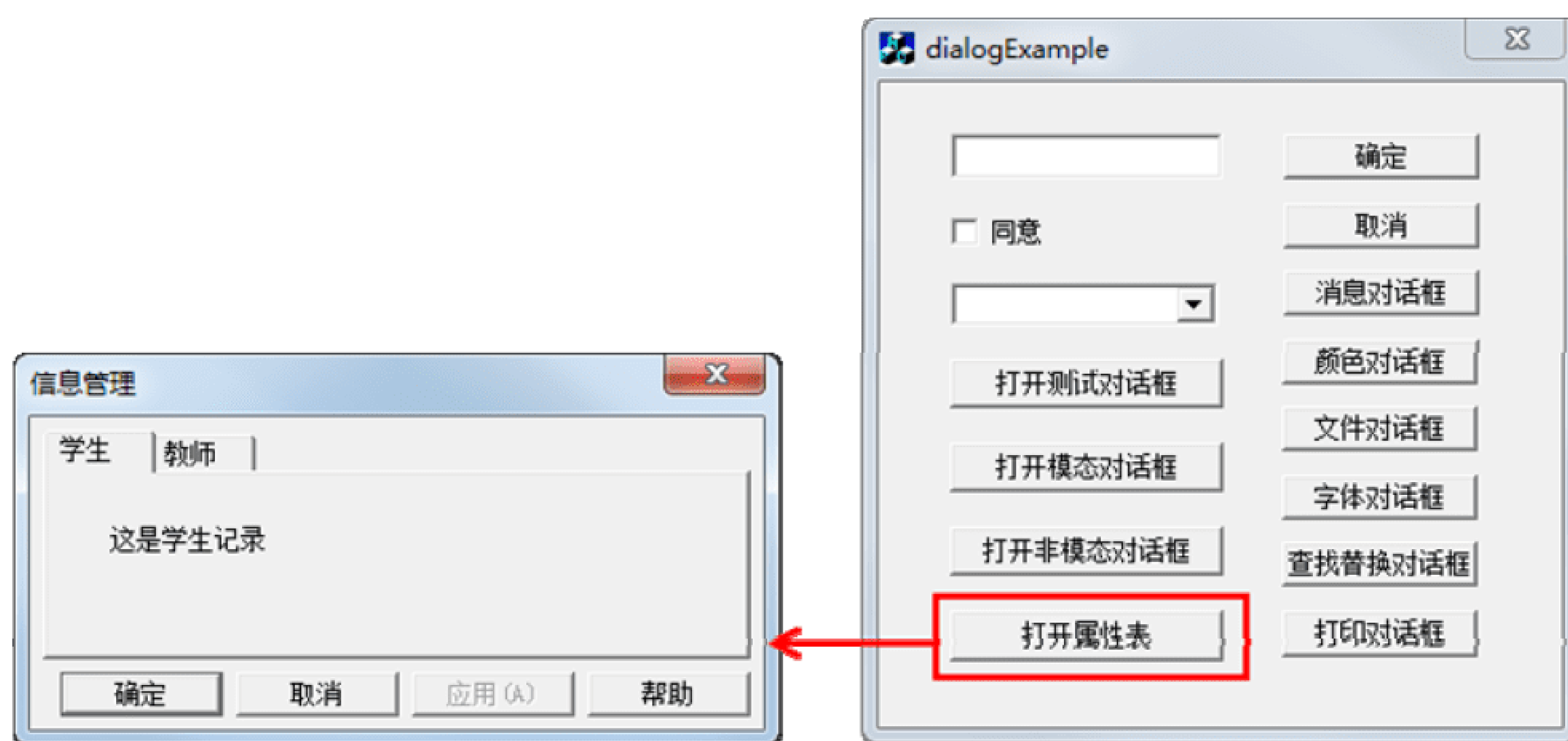


图 10-17　属性表程序效果

（5）根据需要，按照前面介绍过的方法处理属性页和属性表之间的数据交换。

10.5　消息对话框与公用对话框

除了 CDialog 类外，MFC 提供几个从 CDialog 类派生而来的类，封装了常用的对话框功能，这些封装的对话框称为“公用对话框”，是 Windows 公用对话框类的一部分。主要有处理颜色选择的 CColorDialog 类，处理打开和保存文件的 CFileDialog 类，执行查找和

替换操作的 CFindReplaceDialog 类，指定字体的 CFontDialog 类，完成打印工作的 CPrintDialog 类。本节将介绍这些对话框的使用。

10.5.1　消息对话框实例

消息对话框是用于显示提示消息的对话框，是程序用于显示提示信息、错误信息等用户接口的重要组成部分。合理地使用消息对话框可以提高程序的界面友好性。函数原型为：

```
int MessageBox(                          //返回值表示是否成功地显示消息对话框
    LPCTSTR lpszText,                    //要显示的提示信息
    LPCTSTR lpszCaption = NULL,          //要显示的消息对话框的标题
    UINT nType = MB_OK );                //消息对话框的样式
```

在此函数中，通过 nType 参数可以设置消息对话框的样式，由以下几部分组合而成。

（1）指定消息框中包含的按钮，此标记的有效取值如表 10-1 所示。

表 10-1　消息框的按钮标记

标　　记	含　　义
MB_ABORTRETRYIGNORE	消息框中包含 3 个命令按钮：取消、重试和忽略
MB_OK	消息框中包含 1 个命令按钮：确定。此选项是默认选项
MB_OKCANCEL	消息框中包含 2 个命令按钮：确定和取消
MB_RETRYCANCEL	消息框中包含 2 个命令按钮：重试和取消
MB_YESNO	消息框中包含 2 个命令按钮：是和否
MB_YESNOCANCEL	消息框中包含 3 个命令按钮：是、否和取消

（2）指定消息框中显示的图标，此标记的有效取值如表 10-2 所示。

表 10-2　消息框的图标标记

标　　记	含　　义
MB_ICONEXCLAMATION,MB_ICONWARNING	在消息框中显示警告图标
MB_ICONINFORMATION, MB_ICONASTERISK	在消息框中显示带感叹号的图标
MB_ICONQUESTION	在消息框中显示带问号的图标
MB_ICONSTOP, MB_ICONERROR,　MB_ICONHAND	在消息框中显示停止图标

（3）指定消息框中的默认按钮，此标记的有效取值如表 10-3 所示。

表 10-3　消息框的默认按钮取值标记

标　　记	含　　义
MB_DEFBUTTON1	第一个按钮是默认按钮，此选项是默认值
MB_DEFBUTTON2	第二个按钮是默认按钮
MB_DEFBUTTON3	第三个按钮是默认按钮
MB_DEFBUTTON4	第四个按钮是默认按钮

（4）指定消息框的工作方式，此标记的有效取值如表 10-4 所示。

表 10-4　消息框的工作方式标记

标　　记	含　　义
MB_APPLMODAL	要继续程序，必须先对消息框作出响应，此选项是默认选项
MB_SYSTEMMODAL	与 MB_APPLMODAL 标记的作用相同，但是具有 WS_EX_TOPMOST 样式
MB_TASKMODAL	与 MB_APPLMODAL 标记的作用相同，但是使用此标记，当前线程的所有对话框都不可用

以下代码是 MessageBox 的使用示例。

```
01  //消息对话框测试按钮处理函数
02  void CDialogExampleDlg::OnButtonDialogMessage()
03  {
04      MessageBox("Hello World!", "提示", MB_OK);    //显示消息对话框
05  }
```

上面代码使用 MessageBox()函数，弹出标题是“提示”，内容是“Hello World！”的消息框。程序运行效果如图 10-18 所示。

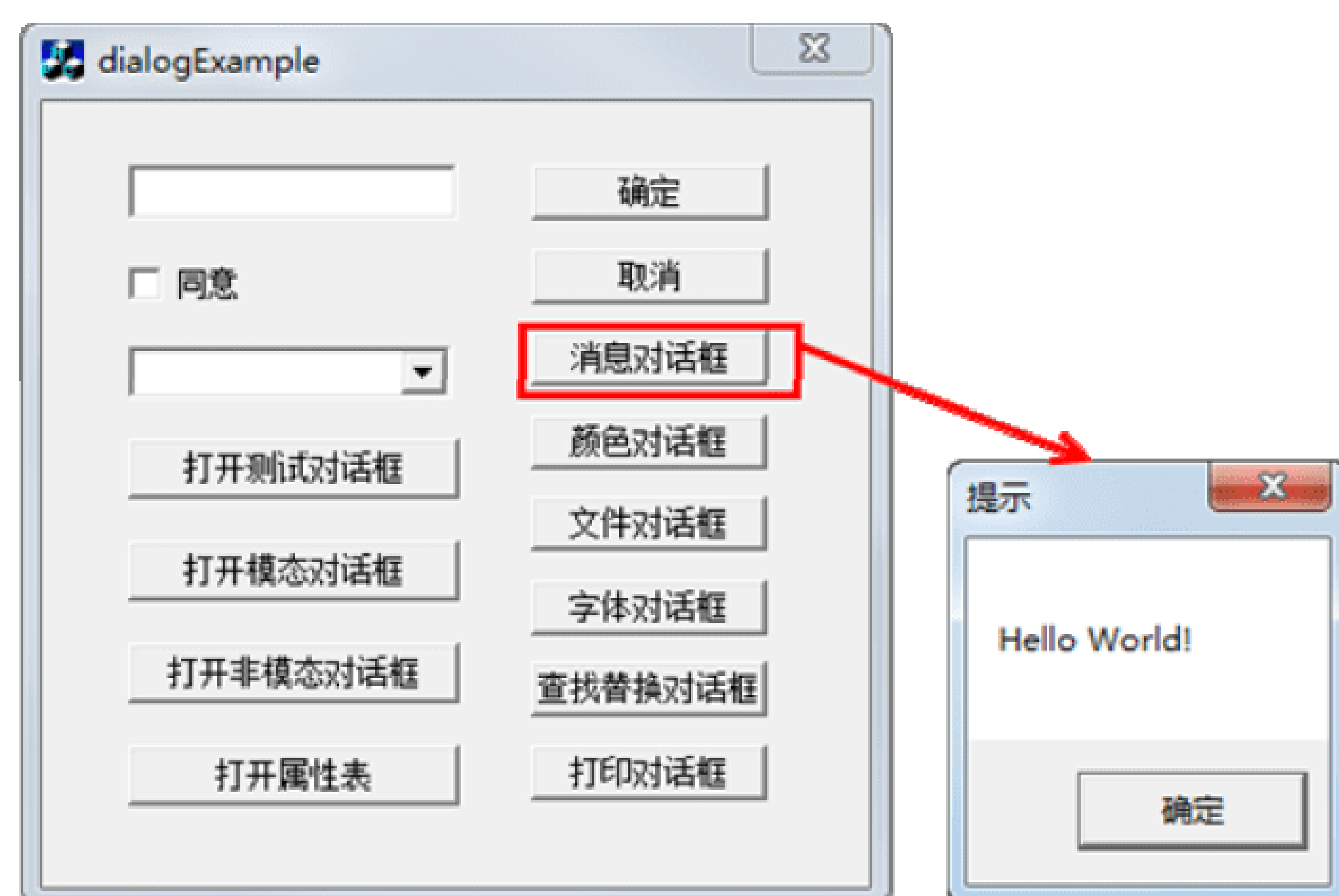


图 10-18　消息对话框示例

10.5.2　颜色对话框实例

在 Windows 程序中，经常会用到颜色选择功能，MFC 提供了 CColorDialog 类实现颜色选择对话框，CColorDialog 对象是有显示系统中定义的颜色列表的对话框。用户可以从列表中选择或创建颜色，当退出对话框时，可以将选择的颜色值返回给应用程序。创建了对话框后，可以设置或修改 m_cc 结构的值以初始化对话框的值。初始化对话框后，调用 DoModal()成员函数显示对话框，并让用户选择颜色。DoModal()函数返回后，通过对话框对象的 GetColor()成员函数可以获取用户选择的颜色。下面的代码显示了颜色对话框的使用。

```
01  CColorDialog dlgColor(m_ctrlCustome);          //初始化颜色对话框
02  if (dlgColor.DoModal() == IDOK)                 //以模态方式显示颜色对话框
03  {
```

```
04      //获取用户从颜色对话框中选择的颜色
05      COLORREF m_ctrlCustome = dlgColor.GetColor();
06  }
```

上面代码首先定义了 CColorDialog 对象，然后调用 DoModal()函数。当函数返回 IDOK 时，根据获取的颜色值执行相应的操作。图 10-19 显示了调用颜色对话框的运行效果。



图 10-19　颜色对话框调用效果

10.5.3　文件对话框实例

CFileDialog 类封装了 Windows 通用文件对话框，提供了完成文件打开和文件保存的简单的方法。此类的样式与 Windows 标准界面是兼容的。读者可以根据自己的需要派生 CFileDialog 类。

要使用 CFileDialog 对象，首先使用 CFileDialog 构造函数创建对象，可以设置或修改 m_ofn 结构的值初始化对话框的值。m_ofn 结构是一个 OPENFILENAME。初始化对话框后，调用 DoModal()成员函数显示对话框，并让用户选择文件。DoModal()函数返回后，通过对话框对象的 GetPathName()成员函数可以获取用户选择的文件的完整路径。下面的代码显示了文件对话框的使用，打开文件对话框，并显示用户选择的文件名。

```
01  void CMyProgram::OnFileButton()          //文件对话框按钮处理函数
02  {
03      //构造文件对话框
04      CFileDialog dlg( TRUE, "EXE", "*.EXE", OFN_FILEMUSTEXIST,
05                       0, this );
06      if ( IDOK != dlg.DoModal() )
07          return;                          //显示文件对话框
08      //在控件中显示用户选择的文件名
09      ((CWnd*)GetDlgItem(IDC_NEW_PROGRAM_NAME))->
10                       SetWindowText(dlg.GetPathName());
11  }
```

上面代码首先定义了 CFileDialog 对象，然后调用 DoModal()函数。当函数返回 IDOK 时，根据选择的文件名执行相应的操作。图 10-20 显示了调用文件对话框的运行效果。

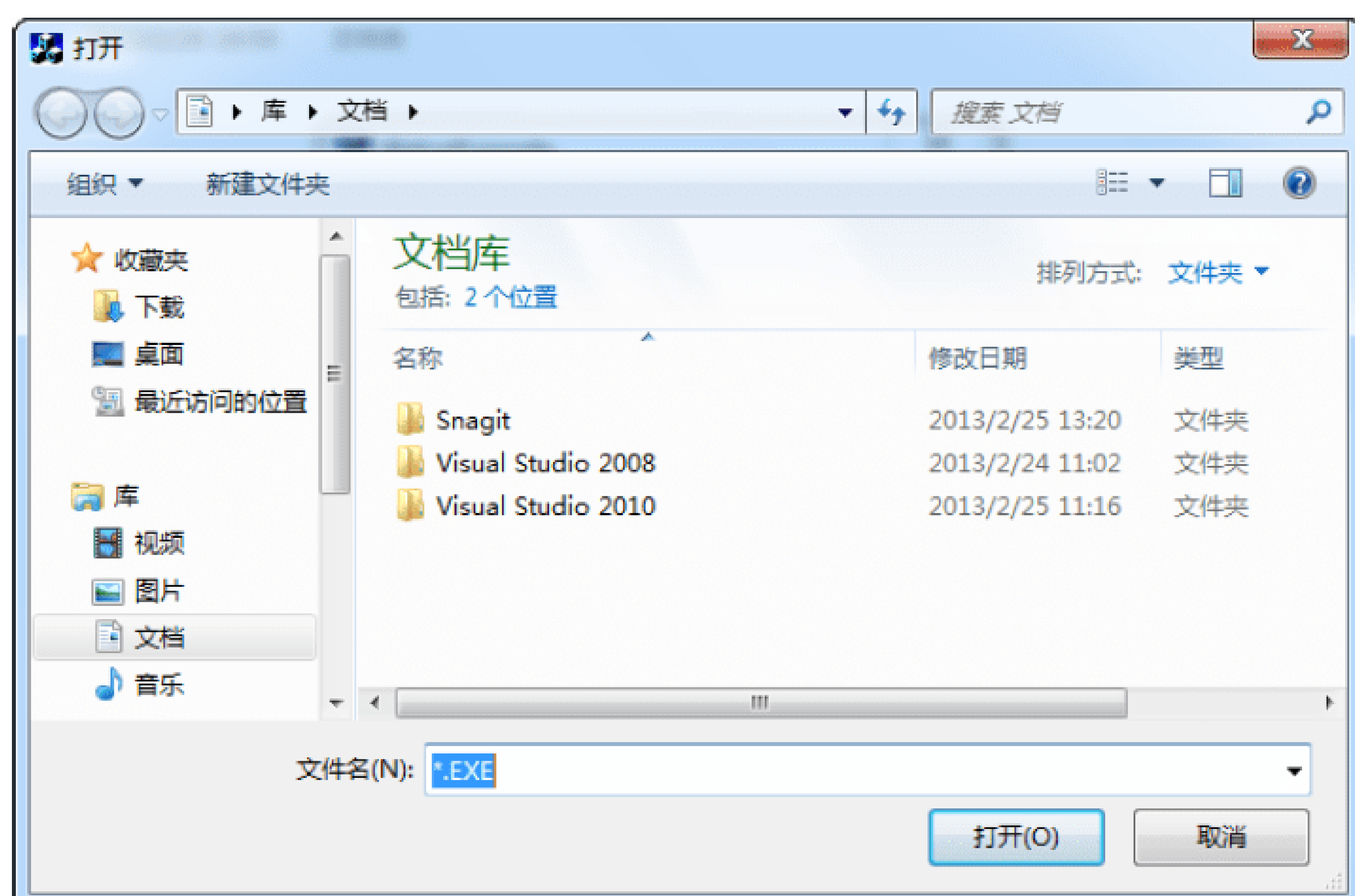


图 10-20　文件对话框调用效果

10.5.4　字体对话框实例

CFontDialog 类允许用户选择字体，其中列出了系统当前安装的字体。用户可以从其中选择特定的字体，并返回给程序。要使用 CFontDialog 对象，首先使用 CFontDialog 构造函数创建对象，可以设置或修改 m_cf 结构的值初始化对话框的值。m_cf 结构是一个 CHOOSEFONT 结构的成员变量。初始化对话框后，调用 DoModal()成员函数显示对话框，并让用户选择需要的字体。DoModal()函数返回后，通过对话框对象的 m_cf 数据成员可以获取用户选择的字体信息。下面的代码显示了字体对话框的使用。

```
01  void CMyProgram::OnSelectFont()              //字体选择对话框
02  {
03       CFontDialog FontDlg;                    //构造字体对话框
04       int ret  = FontDlg.DoModal();           //以模态方式显示字体对话框
05       if (IDOK == ret)                        //如果用户单击了 OK 按钮后
06       {
07            //FontDlg.m_cf 中存放了选择的字体，可以根据需要进行操作
08       }
09  }
```

上面代码首先定义了 CFontDialog 对象，然后调用 DoModal()函数。当函数返回 IDOK 时，根据选择的字体执行相应的操作。图 10-21 显示了调用字体对话框的运行效果。

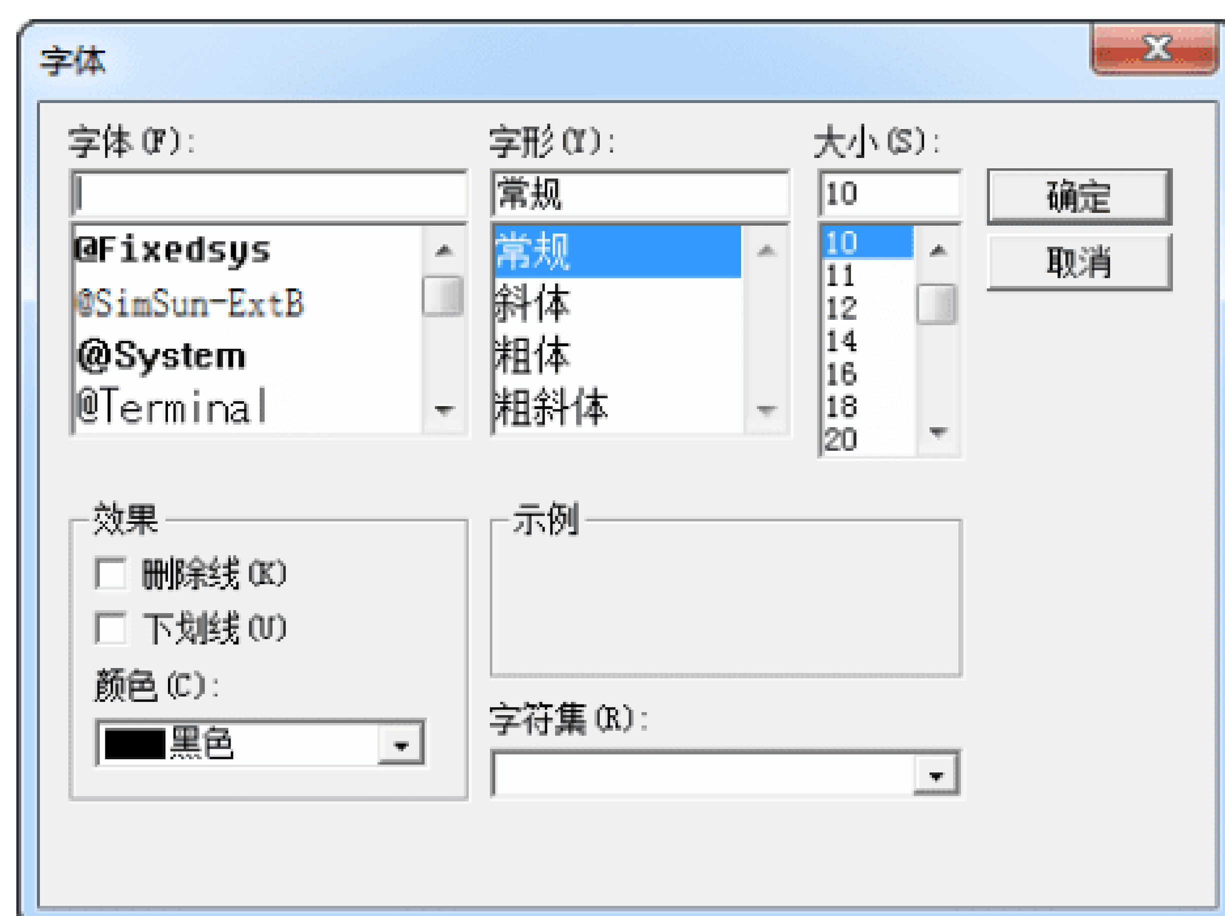


图 10-21　字体对话框调用效果

10.5.5　查找、替换对话框实例

CFindReplaceDialog 类允许用户执行标准的字符串查找替换功能，与普通的 Windows 公用对话框不同的是，此对话框是个非模态对话框，允许用户预期进行交互。CFindReplaceDialog 对话框有两种，一种是查找对话框，一种是查找/替换对话框。要使用 CFindReplaceDialog 对象，首先应使用 CFindReplaceDialog 构造函数创建对象，可以设置或修改 m_fr 结构的值初始化对话框的值。m_fr 结构是一个 FINDREPLACE。初始化对话框后，调用 DoModal()成员函数显示对话框，并让用户执行查找替换操作。下面的代码显示了查找替换对话框的使用。

```
01  //单击查找替换对话框命令
02  void CDialogExampleDlg::OnButtonDialogFind()
03  {
04      CFindReplaceDialog* dlg;                          //定义对话框变量
05      dlg = new CFindReplaceDialog();                   //构造对话框对象
06      dlg->m_fr.lStructSize = sizeof(FINDREPLACE);     //设置查找结构的长度
07      //创建显示查找替换对话框
08      dlg->Create(false, "查找的内容", "替换的内容",
09                  FR_DOWN | FR_WHOLEWORD,     this);
10  }
```

上面代码首先定义了 CFindReplaceDialog 对象，对其进行设置，然后调用 Create()函数。需要注意的是，因为替换查找对话框是非模态对话框，所以必须使用 Create()函数创建显示；同时，要实现真正的查找替换功能需要与相关的视图相连，这里不再赘述。简单的方法是如果创建 CRichEditView 类型的视图，则框架自动完成查找替换功能。图 10-22 显示了调用查找替换对话框的运行效果。

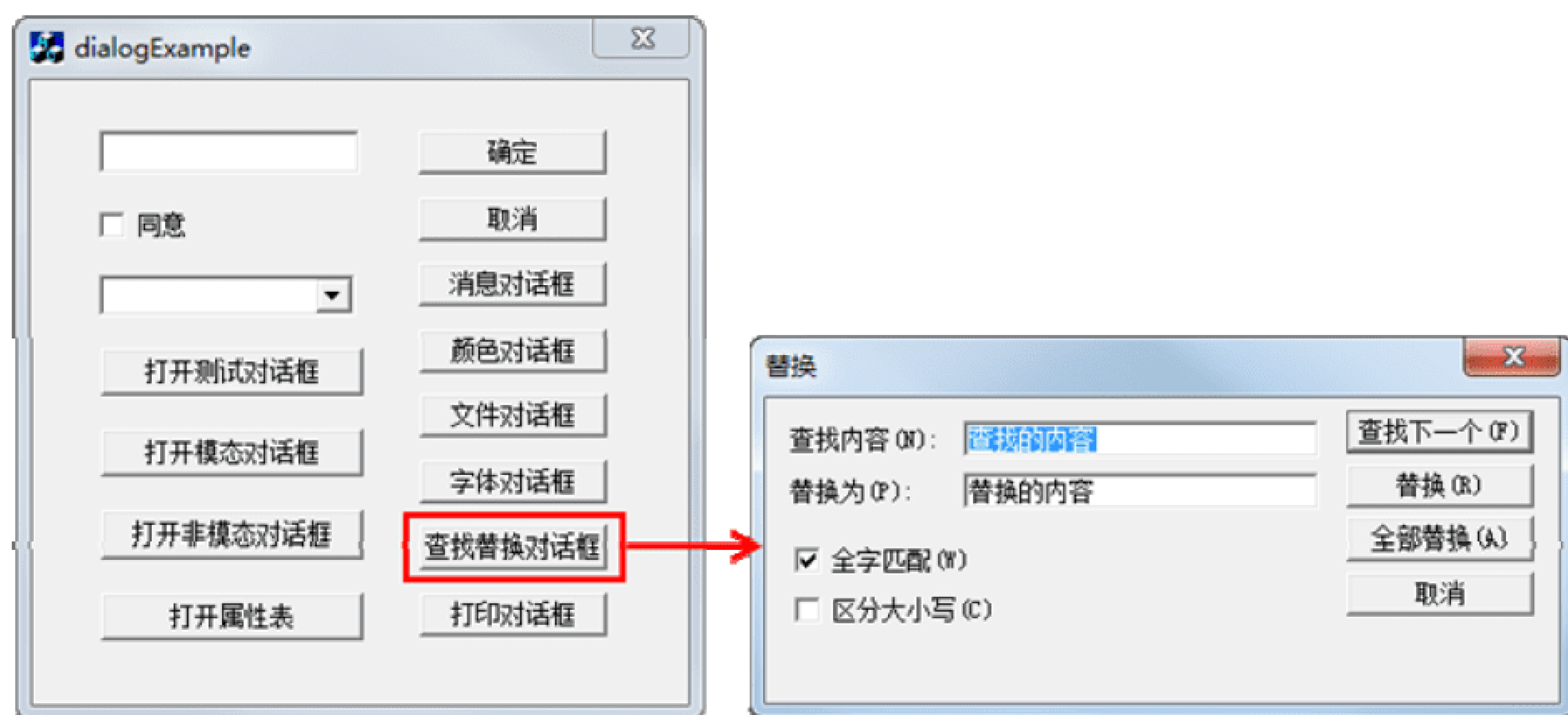


图 10-22　查找替换对话框调用效果

10.5.6　打印对话框实例

CPrintDialog 类封装了 Windows 提供的通用的打印对话框，使用此对话框可以简单地

完成标准的 Windows 打印和打印设置功能。要使用 CPrintDialog 对象，首先使用 CPrintDialog 构造函数创建对象，可以设置或修改 m_pd 结构的值初始化对话框的值。m_pd 结构是一个 PRINTDLG 结构的成员变量。初始化对话框后，调用 DoModal()成员函数显示对话框，并让用户执行打印功能。下面的代码显示了字体对话框的使用。

```
01  void CMyDialog::OnPrintBuf()                    //打印缓冲区数据
02  {
03       char   pbuf[100] = "Hello World.";         //定义信息字符数组
04       HDC    hdcPrn ;                            //定义设备上下文变量
05       CPrintDialog *printDlg =new CPrintDialog(FALSE,
06      PD_ALLPAGES |PD_RETURNDC, NULL);            //构造打印对话框
07      //设置最大页和最小页都为 1
08      printDlg->m_pd.nMinPage = printDlg->m_pd.nMaxPage = 1;
09      //设置打印第一页
10      printDlg->m_pd.nFromPage = printDlg->m_pd.nToPage = 1;
11      printDlg->DoModal();                        //显示打印对话框
12      hdcPrn = printDlg->GetPrinterDC();          //获取使用的打印机句柄
13      //如果选择的打印机句柄为 NULL，则根据需要处理
14      if (hdcPrn != NULL)
15      {
16      }
17      delete printDlg;                            //删除打印对话框
18  }
```

上面代码首先定义了 CPrintDialog 对象，并设置打印对话框的参数，然后调用 DoModal()函数。接着调用 GetPrinterDC()函数判断使用的打印机，并根据情况执行相应的操作。图 10-23 显示了调用打印对话框的运行效果。

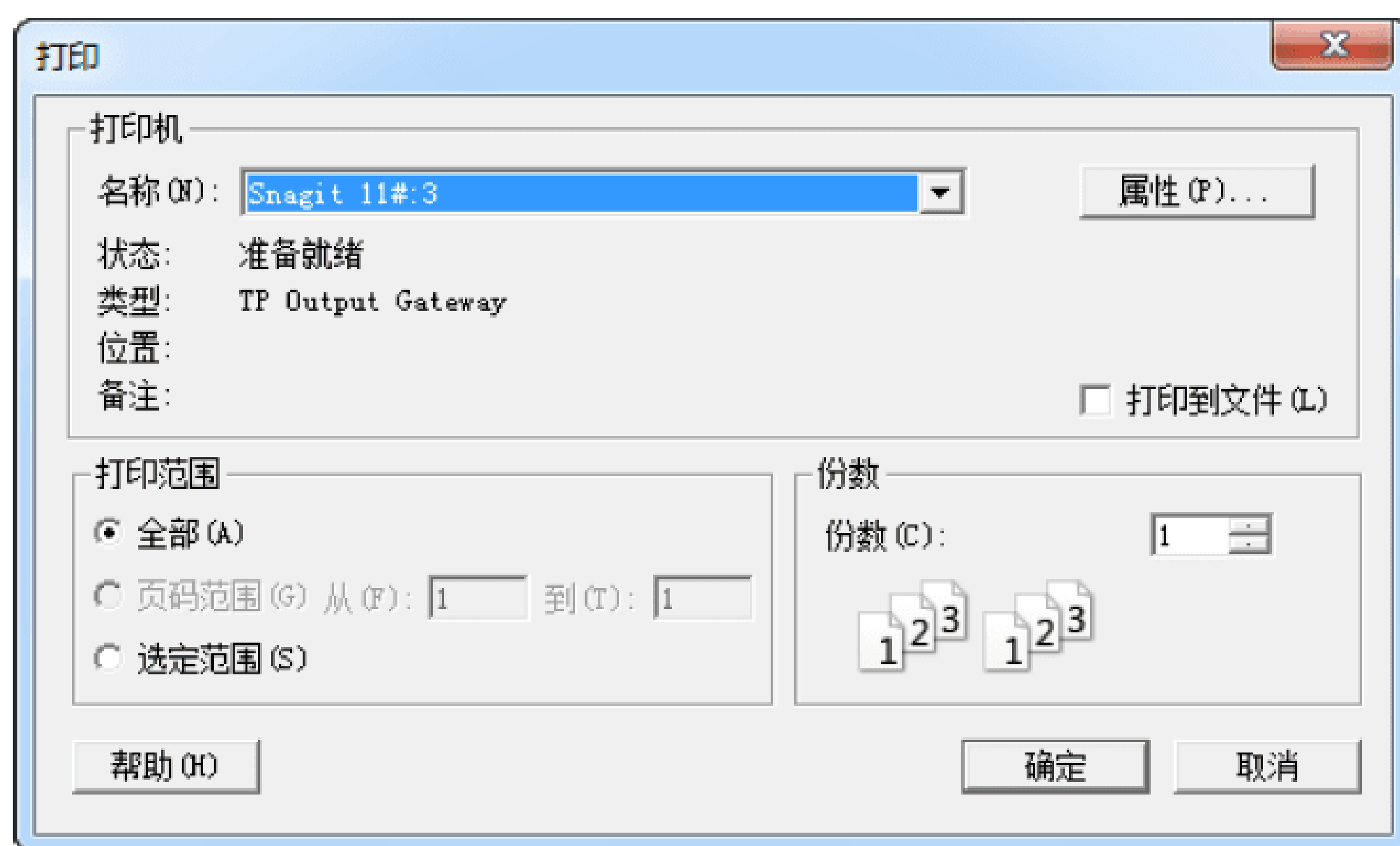


图 10-23　打印对话框调用效果

10.6　本 章 小 结

本章主要介绍了有关对话框的应用。重点讲解了对话框的概念和工作方式，包括对话框数据交换和验证的方式、对话框控件消息的处理以及属性表的使用和公用对话框的使用。本章难点在于掌握各种不同对话框的使用。从第 11 章开始将讲解在 Visual Studio 2010 中如何进行数据库编程。

10.7　习　　题

使用 Visual Studio 2010 提供的模板向导生成的对话框通常会是如图 10-24 所示的样子，单击任意按钮都会关闭对话框，请完成以下操作：

（1）为对话框添加两个控件：编辑框和按钮。使用类向导为编辑框关联一个 int 型的变量，设定取值范围为 0~100；为按钮关联一个 CButton 型的变量。然后参看程序代码中做的改动，体会 DDX 和 DDV 机制。

（2）添加一个按钮“显示消息框”，当被单击时弹出的消息框显示字符串“我的名字是消息框”。

（3）添加一个按钮“打开对话框”，当被单击时弹出“打开”对话框，再单击“打开”对话框上的“打开”按钮时（“打开”对话框见图 10-20），获取选中文件的路径，并用消息框显示出来。

【思路】（1）参考 10.2.3 小节就会很容易地找到相应机制对应的代码。（2）功能类似于 10.5.1 小节的实例。（3）CFileDialog 肯定有获取文件路径的成员函数，可以在 MSDN 中进行查找。

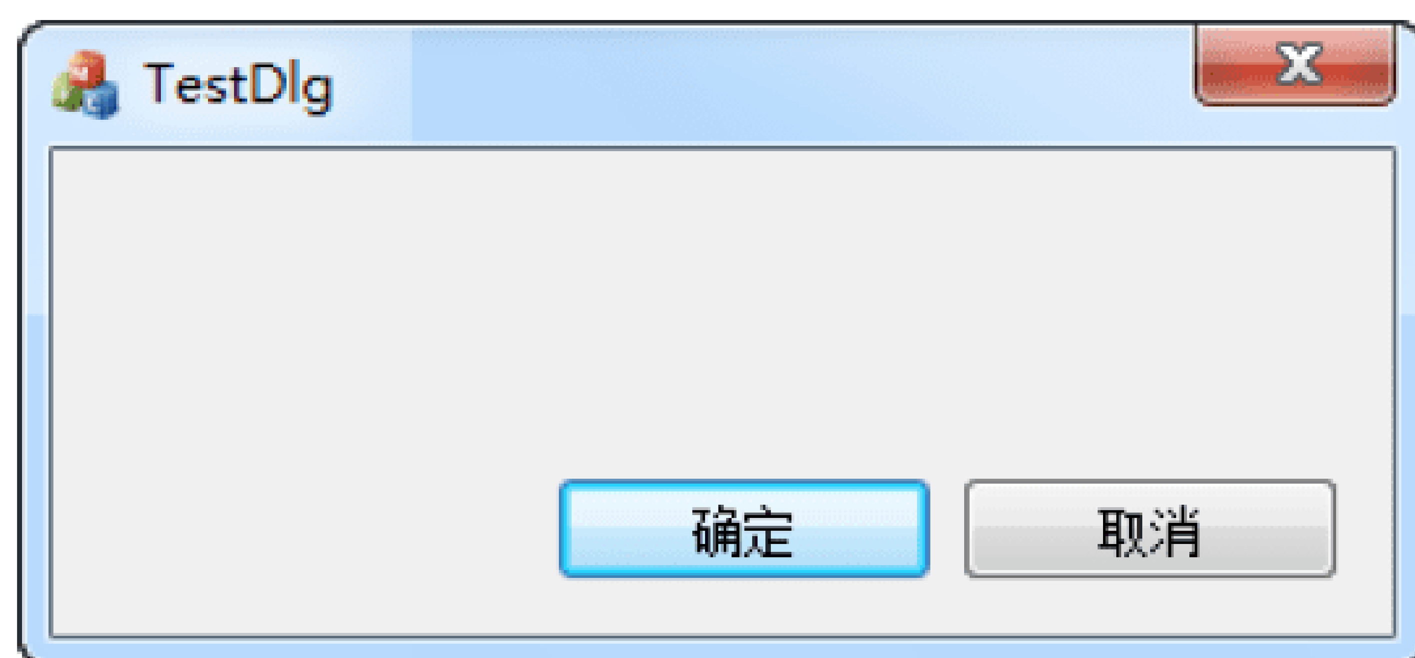


图 10-24　向导生成的对话框

第 3 篇　数据库开发

第 11 章　数据库开发概述

本章主要介绍数据库开发中的基本概念和知识点，使读者对数据库的总体架构和开发过程有一个初步的了解。主要包括数据库简介、数据模型、规范化理论、SQL 语言以及 VC 中的数据库访问接口。

11.1　数据库简介

随着计算机技术的不断发展，对数据存储、编辑、备份和分析的需求越来越迫切，从而引出了数据库技术。简言之，数据库就是存储数据的“库”。近年来，随着数据库应用的深入，人们对通过数据分析帮助决策的需求越来越大。本节首先介绍数据库中的一些基本概念，使读者对数据库有些概念上的认识，这对于后期实际进行数据库编程非常有用。

11.1.1　数据库发展史概述

从 1946 年世界上诞生了第一台计算机开始，至今，数据管理经历了 4 个阶段。

1. 人工管理阶段（20世纪50年代中期以前）

此阶段计算机主要用于科学计算，还没有应用到管理。数据一般不进行保存，没有直接存取的存储设备，也不需要软件对数据进行管理，同时，数据也不是共享的，而是面向应用的。

2. 文件系统阶段（从20世纪50年代后期到20世纪60年代中期）

此阶段计算机不仅应用于科学计算，还开始应用于管理。已经出现了磁盘等直接存取的存储设备，也有了管理数据的软件，也就是文件系统。数据和程序之间通过文件系统的存取接口进行通信。数据和应用之间具有了一定的独立性。

3. 数据库系统阶段（从20世纪60年代中期至今）

随着计算机应用于管理的扩大，应用范围越来越广，数据量越来越庞大，数据共享的要求越来越强，由此产生了数据库。数据库解决了数据与应用之间独立并共享的问题，同时还提供了统一的数据控制功能。从 20 世纪 60 年代中期产生数据库起，到 20 世纪 60 年代末、70 年代初短短的十几年里，数据库技术有了突飞猛进的发展，技术日趋成熟，并形成了坚实的理论基础。这期间，IBM 公司研发了 IMS（Information Management System）层次数据库，开创了数据库的先河。随后，美国数据系统语言协商会，又提出了 DBTG 报

告，为网状数据库模型奠定了技术基础。1970 年，E.F.Codd 发表论文，提出了关系模型的数据库技术。至此，数据库技术在理论基础的指引下，步入了快速发展时期。至此，数据库技术已经非常成熟了。

4．数据仓库阶段（从20世纪90年代至今）

确切地说，数据仓库现在还处在初级阶段，业界还没有对其确切定义。数据仓库之父 Bill Inmon 对其定义为：数据仓库是一个面向主题的、集成的、相对稳定的、反映历史变化的数据集合，用于支持管理决策。数据仓库主要包括数据源、数据存储和管理、OLAP 服务器、前端工具（主要包括报表工具、查询工具、数据分析工具和数据挖掘工具）4 部分，分别是数据的来源、数据的管理、数据的分析和其他面向用户的工具。

目前，正处在数据库系统阶段的快速发展阶段和数据仓库阶段的初级阶段，所以，本书主要研究在 VC 中有关数据库的开发。

11.1.2　数据库常见概念

与数据库有关的基本概念有数据、数据库、数据库管理系统和数据库系统。

- ❑ 数据（Data）是对现实世界事物的抽象。如数字、文字、图像、声音和表格等都是数据。人们将现实世界中的事物进行抽象，定义成数据，并通过数据进行沟通。数据库处理的对象就是数据。
- ❑ 数据库（Database）是存放数据的“库”。它由数据和数据库对象组成。数据库对象指数据库中存储操作数据的对象，如网状数据库中的数据项、记录和系，层次数据库中的字段和片段，关系型数据库中的表、视图、存储过程和触发器等都是数据库对象。
- ❑ 数据库管理系统（DBMS）是管理数据库的系统。它可以实现数据库的创建、数据库的管理、数据库的存取和数据库的维护功能。使用数据库管理系统可以简化数据库设计的工作，提高开发效率。
- ❑ 数据库系统由硬件、操作系统、数据库、数据库管理系统、编译系统、应用程序开发工具、数据库管理员、程序员和用户组成。因此，数据库系统是一个体系结构，是实现数据库应用的所有对象的集合，包括硬件、软件和人员。

11.1.3　数据库的作用

数据库的主要作用就是存储和管理数据。使用数据库具体可以完成下列功能。

- ❑ 结构化存储数据：数据库在存储数据时，是将数据结构化后存储在计算机上的。这样有利于实现数据的查询分析。
- ❑ 使用数据库存储数据，数据冗余少，易扩充。由于数据库是面向多应用的，所以设计合理的存储结构，可以减少数据的冗余。
- ❑ 保护数据的安全性：数据库一般采用验证密码和其他身份验证方式，对数据的存取进行安全控制，以保护数据不被非法读取和泄露。
- ❑ 保护数据的完整性：数据库系统中采用一定的检查机制，保证输入数据的完整性，

进而保证系统的有效性。

- 并发控制：数据库一般都是多用户的，所以在数据库中都对并发做了很好的控制，防止用户并发操作对数据库带来的破坏。

上述数据库的各个作用，在不同的数据库中实现方式都不一样，但是实现思想都是一样的，都是遵循数据库的原理实现的。所以，读者在使用不同数据库时，具体细节需要参考相应的数据库手册，以使得数据库的使用更高效、快速。

11.1.4　数据库管理系统（DBMS）

数据库管理系统（DataBase Manage System，简称 DBMS）是建立在操作系统上，管理和维护数据库的系统。数据库管理系统主要包括以下几个方面的功能。

- 数据库定义功能：DBMS 提供数据定义语言（DDL），定义数据库结构，并将其定义保存在数据字典（又称为系统目录）中。它是 DBMS 存取和管理数据的基础。
- 数据存取功能：DBMS 使用数据操纵语言（Data Manipulation Language，简称 DML）实现对数据库数据的操作，包括查询、插入、编辑和删除。DML 语言分为两类，一类是交互式命令语言，由 DBMS 采用解释执行的方法进行处理；一类是宿主型语言，嵌入高级语言中，DBMS 通过预编译的方法或扩充主语言编译程序的方法处理宿主型语言，再由主语言编译执行。
- 数据库运行维护功能：这也是数据库管理系统的核心功能。主要包括完整性检查、安全性处理、并发控制、存取控制和数据库的内部维护。所有的数据库在数据库管理系统的统一维护下，保证正确有效地运行。
- 数据库管理功能：主要包括数据库初始数据的载入、数据库备份、数据库恢复、数据库性能监视和分析功能等。

由于目前市面上数据库管理系统种类繁多，各个实现也各不相同，所以在使用具体数据库管理系统时，需要根据不同的数据库管理系统采用不同的工作方式。因此，读者应该根据自身的需求，查阅相应数据库管理系统手册。

11.1.5　数据库常见的 4 种数据模型

在计算机中，使用数据模型完成从信息世界到机器世界的转换。数据模型的建模，就是从计算机系统的角度，将概念模型中的概念，抽象成计算机可以识别的数据，从而形成数据模型。目前数据库技术中支持 4 种数据模型，分别是层次数据模型、网状数据模型、关系模型和面向对象模型。

层次模型是符合树数据结构的模型结构。所谓树就是有且仅有一个根结点，并且其他结点只有一个父结点的结构，如图 11-1 所示。

图 11-1 中信息系中有两位教授（教授 A 和教授 B）以及一位助教（助教 A）。每个教授带一个或多个研究生。此结构构成了层次数据模型。支持层次数据模型的数据库称为层次数据库系统。典型的代表是 IBM 公司的 IMS。

网状模型是符合图结构的模型结构。所谓图就是有一个以上的结点，并且其他结点有多个父结点的结构，如图 11-2 所示。

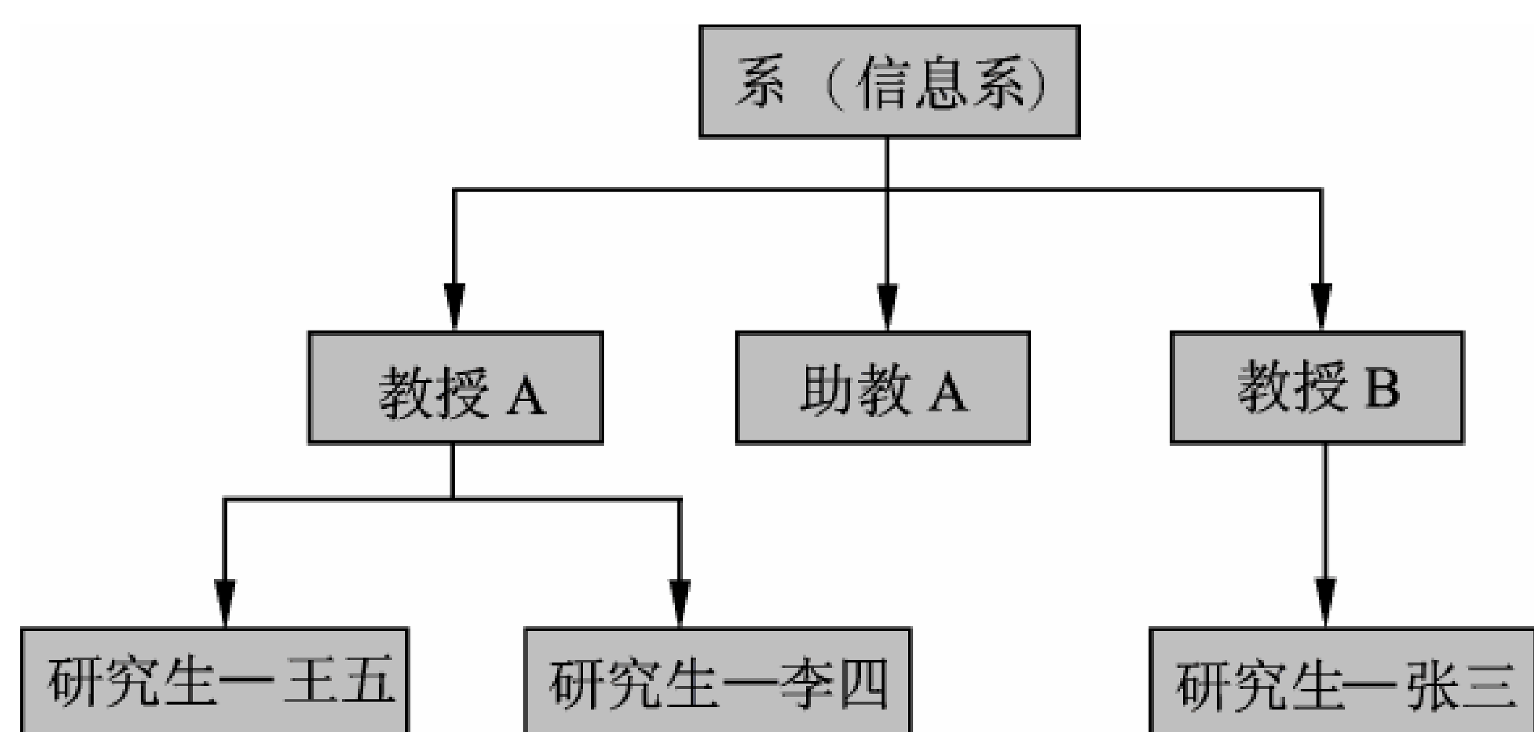


图 11-1　层次模型示意图

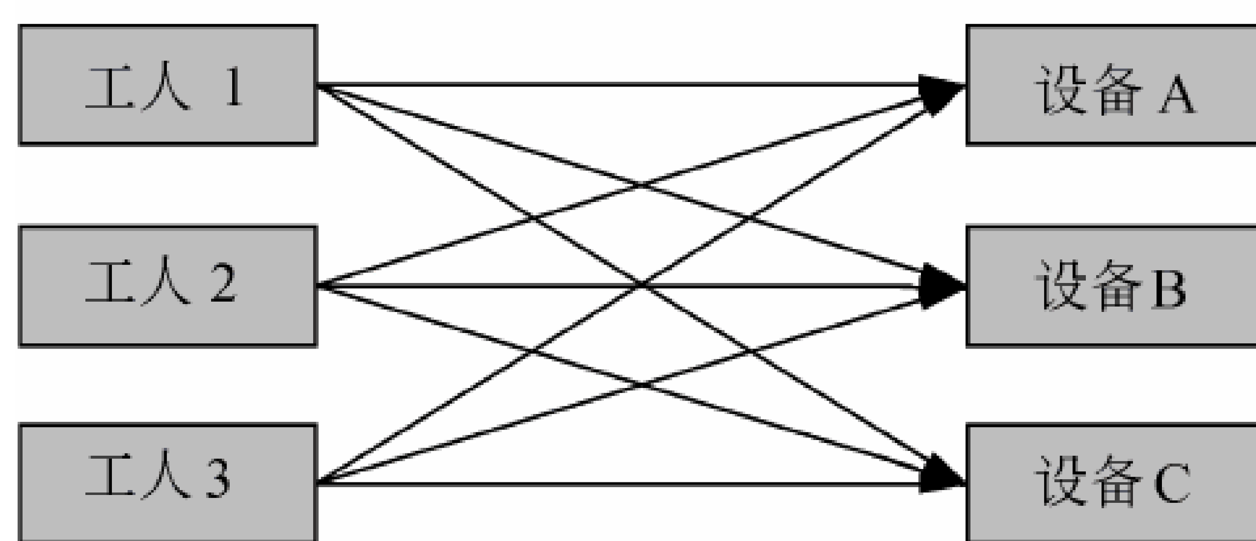


图 11-2　网状模型示意图

在图 11-2 中，工人和设备之间的维修情况形成了网状数据模型的结构，所有的工人可以任意维护一台设备。支持网状数据模型的数据库称为网状数据库系统。网状数据库系统有 IDMS 和 DMS1100 等，其采用的技术基础都是 DBTG 模型。从图 11-2 中还可以看出，网状模型是一个连通的层次模型。也就是说，层次模型可以看作网状模型的一个子集。将层次模型从网状模型中分开，是因为层次模型具有更具体的一些特性。

层次模型和网状模型抽象为数据结构就是图，实体就是图的结点，实体之间的关系就是连接两个结点的线，如图 11-2 中工人 1 就是一个结点，工人 1 对设备 A 的维护关系就是图中对应的连接线。鉴于数据之间的关系的表示，引出了关系模型的概念，而上面两种模型统称为非关系模型，支持这两种数据模型的数据库称为非关系型数据库。

关系模型在逻辑结构上就是一张二维表，用于表示数据之间的关系。自 20 世纪 80 年代以来，数据库厂商推出的数据库产品基本上都是基于关系模型的。很多非关系模型的数据库也提供了关系接口。现在数据库方面的研究也都是基于关系型数据库的。因此，在学习数据库及其编程时，重点应放在关系型数据库上。关系模型中的概念如下所述。

- 关系：一个关系可以看作一张表。
- 元组：也称为记录，表示表中的一行。
- 属性：也称为字段，表示表中的一列，列的名字称为属性名或字段名。
- 主码：也称为关键字，唯一地标识一个元组的属性。
- 域：属性的取值范围。
- 分量：元组中的一个属性值，也称为字段值。
- 关系模式：用关系名（属性名 1，属性名 2，…属性名 n）方式描述关系。

关系数据模型比层次数据模型和网状数据模型结构简单、关系清晰，更有利于对现实世界的抽象和实现。因此，关系数据模型是目前最常用的数据模型。后面介绍的数据库知识都是基于关系型数据库的，图 11-3 所示为关系数据模型的例子。

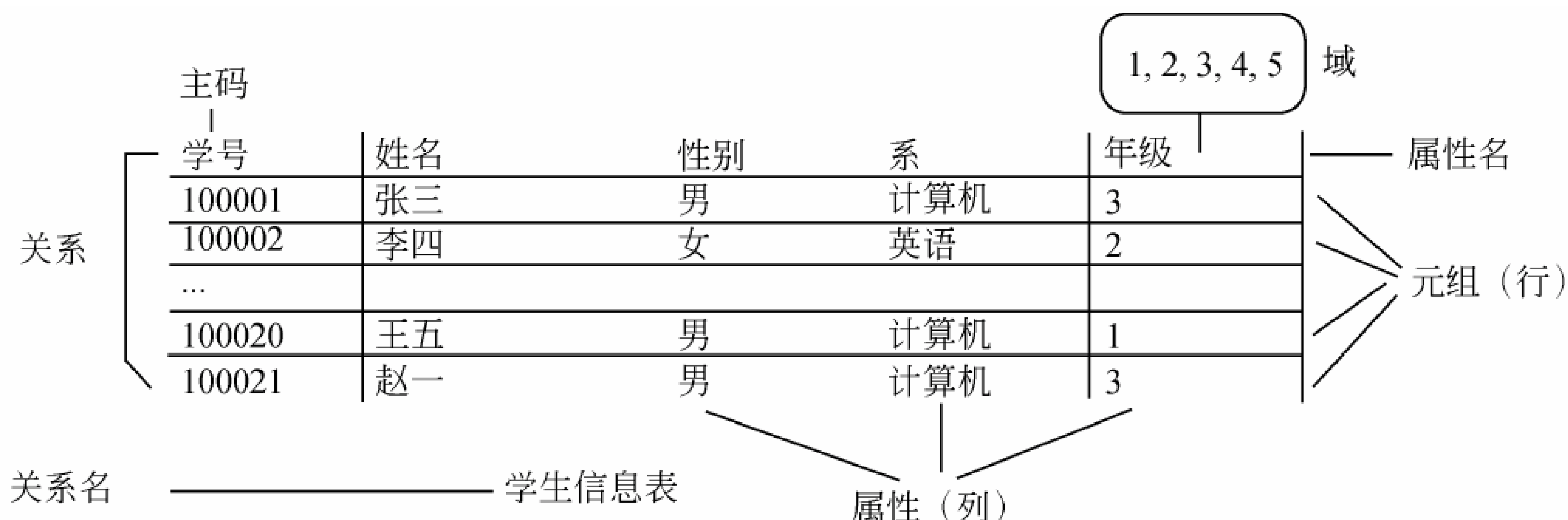


图 11-3　关系数据模型示意图

图 11-3 是学籍管理数据库中的学生信息表，其中学号为唯一标识记录的主码；姓名、性别、系和年级是列；年级属性的取值域为{1，2，3，4，5}。该表用于存储学生信息。

随着面向对象开发技术的发展，出现了面向对象数据模型。面向对象的数据模型将实体作为对象，现实世界的任何实体都可以作为对象看待。与其他对象之间的关系，可以使用属性或方法表示。其模型思维更接近于人类思维，因此，近些年来面向对象模型发展非常迅速，但是还没有达到成熟应用阶段，所以在本书中不做介绍。

11.1.6　数据库的体系结构

数据库体系结构指数据库系统的组成。虽然目前数据库系统各不相同，支持的数据模型不同，使用的数据库语言不同，运行在不同的操作系统上，数据存储的方式也不相同，但是大部分数据库系统体系结构都是相同的，都是符合三级模式的。数据库系统的三级模式结构由外模式、模式和内模式组成。

- 外模式：也称为子模式或用户模式，是数据库用户看到的数据视图，它是呈现给应用的数据集合。
- 模式：也称为逻辑模式，是数据在数据库中的逻辑结构和描述，是所有用户使用的公共数据视图，不会根据用户的不同发生变化。
- 内模式：也称为存储模式，是数据在数据库中的内部表示，即数据的物理结构和存储方式。它是独立于用户数据视图的。如内模式需要确定记录是按照顺序存储还是链式存储，索引的组织方式，数据在存储时是否加密，使用什么加密方式等。

数据库的三级模式，将数据的存储方式、逻辑结构和用户应用视图分开，使得用户不需要关心数据的具体存储方式，只需要关心业务逻辑就可以了，这样简化了数据库开发人员的工作量。如图 11-4 所示，显示了数据库系统的体系结构。

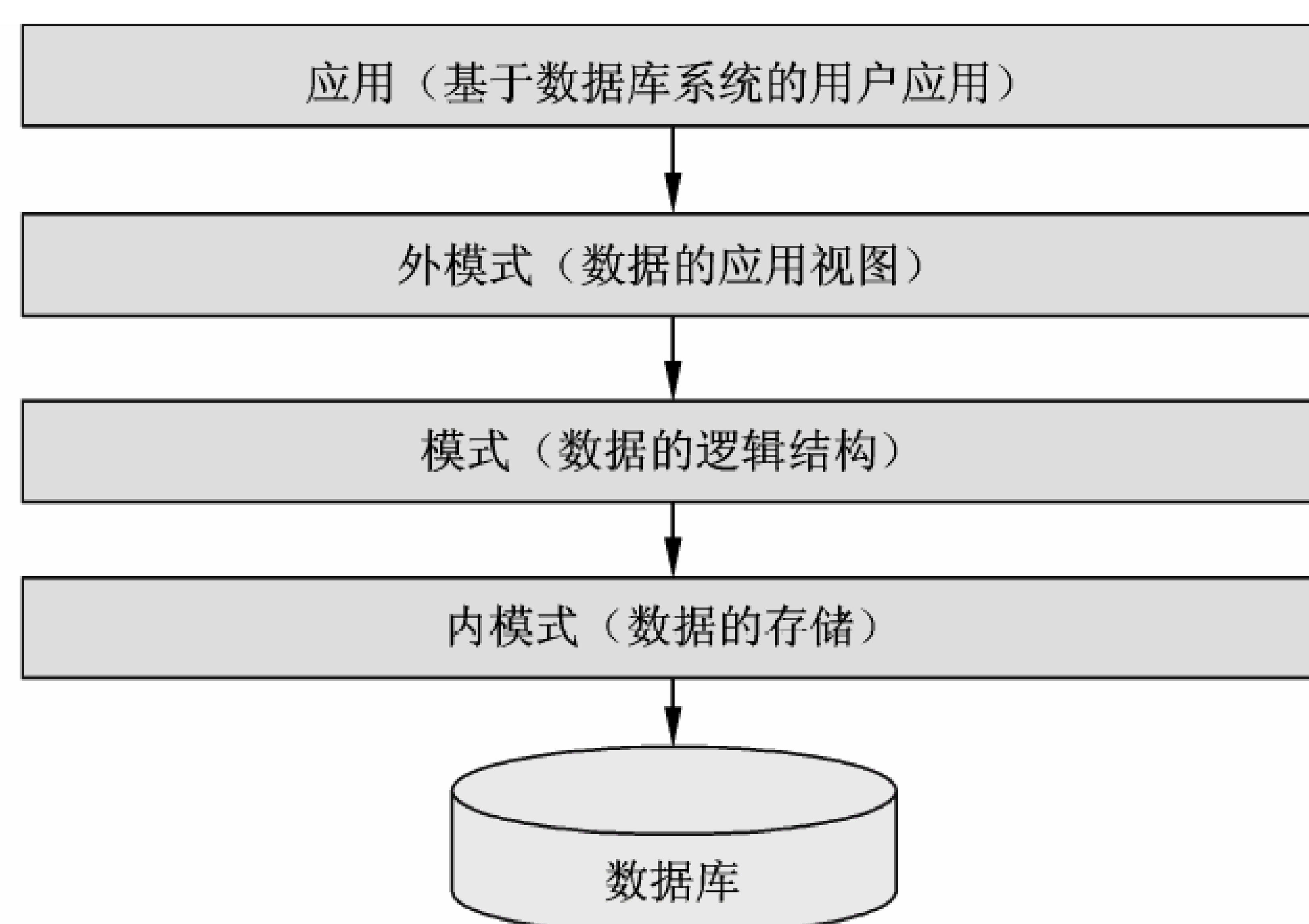


图 11-4　数据库系统的体系结构

11.1.7　关系数据库

前面介绍过数据模型，其中提到了最常用的关系模型，基于关系模型的数据库称为关

系数据库。下面是与关系数据库相关的概念。

- 表：关系数据库使用二维表格存储数据，类似于工作表格，是将数据按照行和列相结合的方式存储。一个数据库中可以包含多个表。如学生表存储学生信息，教师表存储教师信息。
- 字段：关系表中的每一列称为字段。表是由多个字段共同组成的，每个字段表示某个方面的含义。如学生表中姓名字段用于表示学生的姓名。
- 记录：关系表中的每一行称为记录。如学生表中每条记录存储一个学生的信息。
- 主键：用于唯一标识每条记录的一个或多个字段的组合。表中任意两条记录的主键不可以重复。如学生表中使用学号作为主键，每个学生的学号都是不相同的，而姓名不能作为主键，因为学生是有可能有重名的。再如学生成绩表中，使用学号加课程代码组合键作为主键，只有这两者的组合键才能唯一区别记录。
- 索引：表中单列或多列数据的排序列表。主要用在数据查询中，可以提高查询速度。如要根据学号检索学生信息，可以在学号上建立索引。
- 关系：数据库中的各个表之间存在一定的关系，通过关系将各个表之间的数据联系起来。如要查看每个学生的姓名和各科成绩，则需要通过学号将学生表和学生成绩表关联起来。

11.1.8　数据库的开发过程

基于数据库的开发是一个复杂的过程，这个过程需要经历 3 个阶段。

1．从现实世界到信息世界

现实世界客观存在的图表、数字和声音等事物就是机器世界要处理的原始数据，而计算机只能处理数据，所以，第一步需要设计人员根据用户的需求将现实世界的原始数据转换到信息世界，进行数据建模，这一步也称为概念模型的建模。此步骤可以建立程序人员和用户都可以理解的概念模型。11.3 节就将介绍概念模型的建模方法 E-R 模型。

2．从信息世界到机器世界

数据从信息世界到机器世界的建模，就是将概念模型中的数据转换成机器可以识别的数据，这一步也称为数据模型的建模。当数据从概念模型中转换到数据模型后，就变成了计算机可以识别的模型。

3．在机器上实现应用

数据模型建好后，系统开发人员就可以根据建好的数据模型，在计算机上实现相应的数据模型，这时就需要相应模型的数据库的支持，在相应的数据库上设计符合需求的数据库。在后面 11.2 节将介绍关系数据库的设计理论——规范化理论，在 11.4 节中将介绍关系数据库语言——结构化查询语言（SQL），在 11.5 节中将介绍 VC 提供的数据库接口。

11.2　规范化理论

数据库解决了数据的存储问题，实现了数据的共享，提高了开发效率。但是要提高数据库的使用效率，需要合理地设计数据库结构，这就要求数据库设计人员遵循一定的原则。其中，E.F.Codd 于 1971 年提出的数据库规范化理论是比较重要的一种数据库设计理论。最早规范化理论是针对关系模型数据库设计的，但是其对其他模型的数据库设计也具有指导意义。在本书中主要介绍关系型数据库，所以，以规范化理论在关系型数据库中的应用为例，介绍规范化理论。

11.2.1　为什么需要规范化

前面对关系型数据库做了简单介绍。下面看一个关系型数据库的设计例子，以供应商供货信息为例，如下所示：

```
供应商供货信息(供应商公司名称，供应商联系人，供应商地址，供应的产品名称，供应的产品数量)
```

上面定义了一个供应商关系模式，属性分别是供应商公司名称、供应商联系人、供应商地址、供应的产品名称和供应的产品数量。供应商供应一种货物，则对应上述关系模式中的一条记录。根据现实生活的实际情况，会发现这个关系模式存在以下弊端。

- 数据冗余：当供应商每供应一种货物，则需要插入供应商公司名称、供应商联系人、供应商地址、供应的产品名称和产品单价。这样，当一个供应商供应超过一种货物时，则记录中会出现多次的供应商公司名称、供应商联系人、供应商地址等属性值；同时，当一种货物更换供应商供应时，记录中会出现多次同样的产品名称。这样，随着业务的进行，数据量越来越大，则冗余数据会越来越多。
- 插入错误：当一个供应商暂时没有供应任何商品时，无法记录供应商的信息，包括供应商公司名称、供应商联系人和供应商地址。
- 更新错误：当供应商的地址发生变化时，由于数据冗余，有可能出现在一条记录中更改了供应商地址，而在另一条记录中没有更改供应商地址的情况。这样，破坏了数据的一致性，与实际情况不符。
- 删除错误：当供应商供应的所有货物被删除后，供应商的信息也被删除了，则不能保存曾经供应过货物的供应商的信息，遗漏了实际情况中的信息。

从上面可以看出，这个关系模式存在许多弊端，造成了数据冗余，同时有可能破坏数据的一致性和完整性，因此，这个关系模式的设计是不合理的。鉴于上面这种情况，加之数据库是对现实世界的抽象，人们就需要研究关系模式设计的原则。根据这些原则可以设计出既能满足关系模式思想，又在数据处理上更合理的数据库，由此产生了规范化理论。下面将介绍有关规范化的理论知识。

11.2.2　数据依赖

从 11.2.1 小节的例子中可以看出，供应商联系人和供应商地址都是由供应商决定的，

也就是说，一旦供应商确定了，则供应商联系人和供应商地址就确定下来了，这种关系就是一种数据依赖，即一个属性的值依赖于另一组属性的取值。数据依赖是现实世界事物之间联系的抽象，表示关系模式中任何一个关系取值都必须满足一种约束条件，这种约束条件是通过关系中属性值的相等与否确定的。数据依赖的种类有很多，其中最重要的就是函数依赖和多值依赖。

函数依赖是指在关系模式 R 下，任何可能的关系 R 都必须满足：R 中不可能有在两个元组中一组属性值 X 相等，而在另一组属性 Y 不相等的情况。并且称这种情况为 X 函数决定 Y，或者称为 Y 函数依赖于 X，记作 X->Y。

- X->Y，但是 X 函数不依赖于 Y，则称 X->Y 是非平凡的函数依赖。默认情况下，都是指非平凡的函数依赖。
- 若 X->Y，则 X 叫做决定因素。
- 若 X->Y，Y->X，则记作 X<->Y。
- 若 Y 不函数依赖于 X，则记作 X-/->Y。
- 若 X->Y，并且对于 X 的任何一个真子集 X’，都有 X’->Y，则称 Y 对 X 完全函数依赖，记作 X（f）->Y。

函数依赖是针对一个关系模式下的所有可能的关系取值都必须满足的条件，并且这种依赖必须在任何时刻都成立，才可以称为关系模式满足函数依赖。

如上面的供应商例子，根据函数依赖的概念，分解为以下两个关系模式：

```
供应商信息（供应商公司名称，供应商联系人，供应商地址）
供应商供货信息（供应商公司名称，供应的产品名称，供应的产品数量）
```

其中在供应商信息关系模式中，包括供应商公司名称->供应商联系人，供应商公司名称->供应商地址。函数依赖是一个语义范畴的概念。如在这个关系模式中，假定不允许存在同名的供应商，如果存在同名的供应商，则会出现相同的供应商公司名称，出现不同的供应商联系人和供应商地址，所以也就不符合供应商公司名称->供应商联系人的函数依赖了。因此，这里所说的函数依赖，是设计者在设计时，对现实世界作了强制规定。在这里是指，强制规定不允许存在同名的供应商。由此引出了码的概念。

既然规定了不允许存在同名的供应商，就形成了供应商公司名称函数依赖于整个供应商信息属性组，这种情况下称供应商公司名称为供应商信息关系模式的候选码（Candidate key）。一个属性组中可能出现多个候选码，选择其中一个作为标识，称为主码（Primary key）。包含主码的属性称为主属性（Primary attribute）。不包含在任何候选码中的属性，称为非主属性（Non-primary attribute）或非码属性（Non-key attribute）。在实际情况中，有可能使用单个属性作为码，如供应商公司名称就是供应商信息的码；也可能出现多个属性组成属性组作为码，如上面的供应商供货信息中的（供应商公司名称，供应的产品名称）属性组是码；也有可能使用整个属性组作为码，称这种情况为全码，如选课信息的关系模式。如下所示，其中整个属性组是其码。

```
选课信息（学生姓名，教师姓名，课程名称）
```

关系模式最核心的思想是将实体和实体间的关系都表示为关系。码的概念可以说是将实体本身表示为关系。关系模型中使用外码表示实体间的关系。如上面的供应商例子中，供应商公司名称不是供应商供货信息关系模式的码，但它是供应商信息关系模式的码，这

种情况下，称供应商公司是供应商供货信息关系模式的外码（Foreign key）。

除了函数依赖外，还有一种比较重要的数据依赖，就是多值依赖。如关系模式仓库保管员保管物品（仓库、保管员和物品）。假设，每个仓库有多个保管员，保存多种物品，每种物品由所在仓库的所有保管员保管，如表 11-1 所示。

表 11-1　仓库保管员保管物品关系模式

仓　　库	保　管　员	物　　品
仓库 A	张三 李四 王五	工作服 工作手套 雨鞋
仓库 B	李四 钱六	锤子 钳子 刀子

这个关系模式的数据对应成二维表，如表 11-2 所示。

表 11-2　仓库保管员保管物品数据对应的二维表

仓　　库	保　管　员	物　　品
仓库 A	张三	工作服
仓库 A	张三	工作手套
仓库 A	张三	雨鞋
仓库 A	李四	工作服
仓库 A	李四	工作手套
仓库 A	李四	雨鞋
仓库 A	王五	工作服
仓库 A	王五	工作手套
仓库 A	王五	雨鞋
仓库 B	李四	锤子
仓库 B	李四	钳子
仓库 B	李四	刀子
仓库 B	钱六	锤子
仓库 B	钱六	钳子
仓库 B	钱六	刀子

由表 11-1 和表 11-2 可以看出，虽然这个关系模式符合函数依赖，但是，其中有许多的冗余数据。在每次增加一个仓库保管员时，都需要增加与此仓库中保管的物品个数一样的多条记录，造成了数据的冗余，同时为后期数据的维护带来困难。这个关系模式中就存在多值依赖。所谓多值依赖，就是假设关系模式 R、U 是属性集，X、Y 是 U 的子集，并且 Z 是 X 和 Y 在 U 上的子集，即 X+Y+Z=U。假如，R 关系模式上的任意关系 R，给定的一组（X，Z）值，Y 的值仅与 X 的值有关，而与 Z 的取值无关，这种情况下称为 Y 多值依赖于 X。如上面的例子中，仓库保管员是多值依赖于仓库的。

11.2.3　范式介绍

在设计关系模式时，需要满足一定的条件。根据满足的条件不同，称为符合不同的范

式。范式从低到高，满足高一级的范式总是同时满足低一级的范式，如满足第二范式的关系模型则一定满足第一范式。根据范式的级别，依次有第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、Boyce-Codd 范式（BCNF）和第四范式（4NF）。第几范式表示关系模式满足一定条件，称为关系模式是符合第几范式的关系模式。为了方便，使用 R∈2NF 表示关系模式 R 符合第二范式。下面分别介绍这几种范式。

第一范式要求关系模式必须满足：元组中的每个分量必须是不可再分的数据项。第一范式是最基本的规范化。

若关系模式 R∈1NF，并且每个非主属性完全依赖于码，则关系模式 R 满足第二范式。也就是说，符合第二范式的关系模式，不允许有非主属性对码的部分函数依赖。如上面供应商供货信息的关系模型。在这个关系模型中，可以看出主码是（供应商公司名称，供应的产品名称）。而供应商地址是部分依赖于主码的，因为它完全依赖于属性组（供应商公司名称）。因此，供应商供货信息关系模型不符合第二范式，这就造成前面小节中描述的问题。因此需要使用分解的方法，将这个关系模型分解成多个符合第二范式的关系模式。如上面这个例子可以分解为下面两个关系模型，其中供应商信息关系模型的主码为供应商公司名称，供应商供货信息关系模型的主码为（供应商公司名称，供应的产品名称）。这样这两个关系模型就都符合第二范式了。

```
供应商信息（供应商公司名称，供应商联系人，供应商地址）
供应商供货信息（供应商公司名称，供应的产品名称，供应的产品数量）
```

若关系模式 R∈2NF，并且每个非主属性不传递依赖于码，则关系模式 R 满足第三范式。也就是说，符合第三范式的关系模式，不允许有非主属性传递依赖于码。如下面的关系模型：

```
员工信息（员工编号，员工名称，员工性别，员工部门编号，员工部门名称）
```

员工信息这个关系模式，存在员工编号->员工部门编号，员工部门编号->员工部门名称，因此员工部门名称传递依赖于员工编号，所以这个关系模型不符合第三范式。不符合第三范式的关系模式，也存在上述的数据冗余、插入错误、更新错误及删除错误的问题，因此需要分解这个关系模式，使其符合第三范式。如下所示，这样就符合三范式了。

```
员工信息（员工编号，员工名称，员工性别，员工部门编号）
部门信息（部门编号，部门名称）
```

若关系模式 R∈1NF，并且每个函数依赖 X->Y,X 中都包含码，则称关系模式 R 符合 BCNF 范式。BCNF 范式是第三范式的修正，有时也称为第三范式。如：

```
学生选课（学生，教师，课程）
```

学生选课关系模式中，假定一个学生可以选修多门课程，一门课程可以由多个学生选修，一名教师只能教授一门课程，一门课程可以有多名教师教授。候选码有（学生，课程）和（学生，教师），因为没有任何非主属性对码传递依赖或部分依赖，所以学生选课关系模式符合第三范式，但是教师->课程，而教师不包含码，所以这个关系模式不符合 BCNF。不符合 BCNF 的关系模式，也会带来插入错误、删除错误等问题，所以，也需要分解关系模式，使其符合 BCNF。上面的学生选课关系模式可以分解为教师课程和学生选课两个关

系模式，则符合 BCNF 范式。

```
教师课程（教师，课程）
学生选课（学生，教师）
```

上面讲述的 1NF、2NF、3NF 和 BCNF 都是基于函数依赖范畴的，如果一个关系模式符合 BCNF，则在函数依赖范畴内，模式设计就是合理的。

前面介绍过多值依赖的概念，现在再结合上面多值依赖的例子关系模式仓库保管员保管物品（仓库、保管员和物品）。为了解决多值依赖的问题，可以将其分解为：

```
仓库物品（仓库，物品）
仓库保管员（仓库，保管员）
```

这样就避免了多值依赖的问题。由此，可以看出多值依赖实际上是属性之间存在非平凡的非函数依赖。因此，4NF 的核心就是消除属性之间的非平凡且非函数依赖的多值依赖。若关系 R∈1NF，并且对于每个非平凡的多值依赖，X 中都包含码，则 R∈4NF。虽然 4NF 与 3NF 的理论不同，4NF 是基于多值依赖概念的，3NF 是基于函数依赖概念的，但是可以证明如果关系模式符合 4NF，则一定符合 3NF。

第五范式的原则是将表尽可能的分割成更小的表，消除表中所有的冗余。由于在实际操作中一般数据库设计到符合 4NF 就可以了，所以，这里不对第五范式做详细讲述了。

本小节主要介绍了数据库的规范化设计，在实际的数据库操作中，至少需要满足 BCNF 范式，所以在进行数据库设计前，需要读者深入了解范式的概念。本小节内容总结如图 11-5 所示。

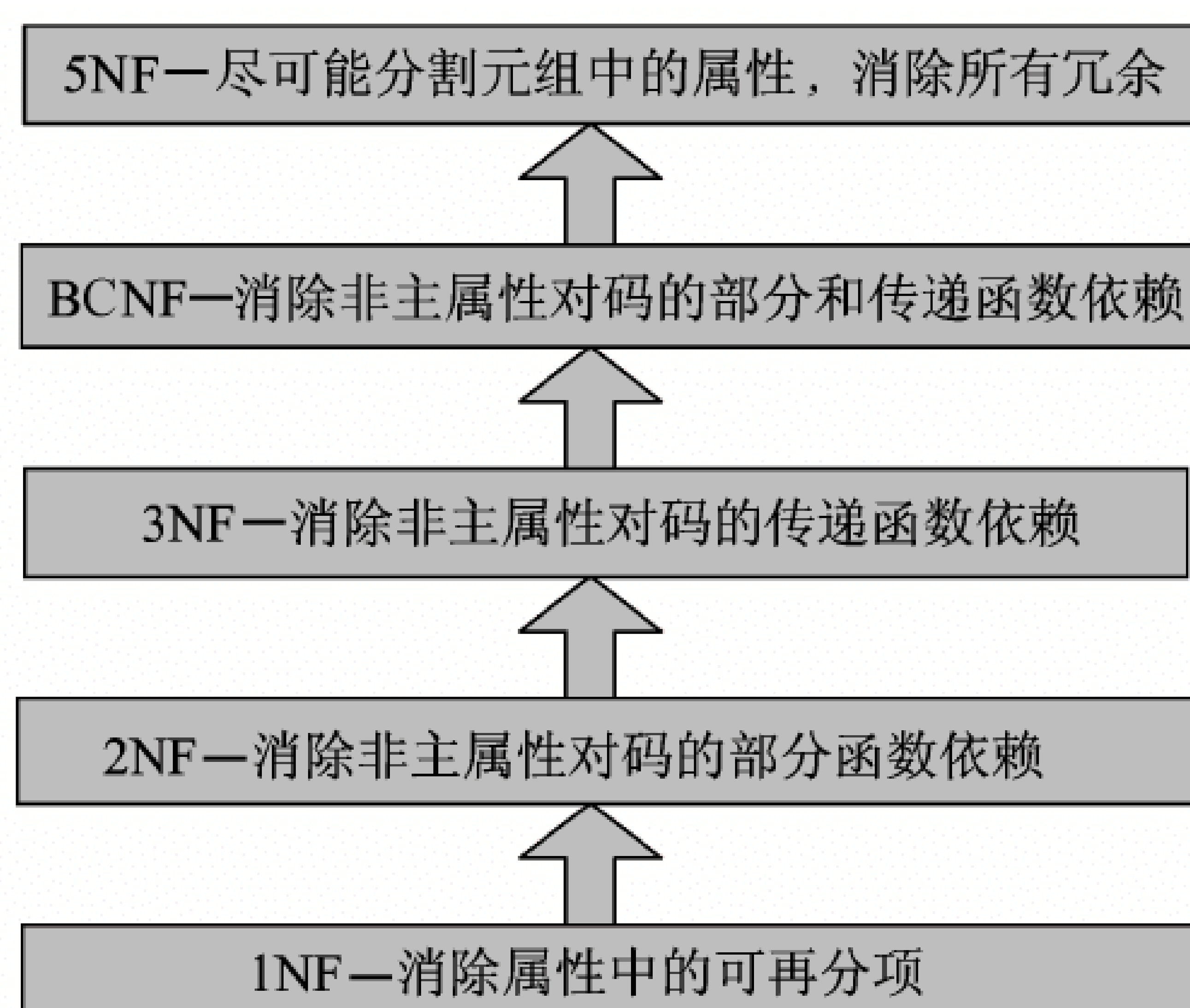


图 11-5　范式的功能

规范化的核心思想就是消除关系模式中的数据依赖部分不合适的部分，使关系模式达到合理的分离，使得一个关系描述一个实体或者一种联系，把一个概念中的多个实体和实体间的联系分离成单个实体或实体联系，使得概念单一化。

11.3　E-R 模型

在数据库设计中，从现实世界到信息世界的第一层抽象过程的成果是概念模型，也叫做信息模型。建立概念模型的典型方法是 E-R 模型，即实体-联系模型。本节将主要介绍

E-R 模型的基本概念和设计方法。

11.3.1　E-R 模型元素

数据库的设计需要将用户的需求从现实世界抽象到信息世界中，成果是概念模型，也叫做信息模型，是按用户的观点，对数据和信息进行建模，并为后期数据库设计人员提供数据模型的基础。因为思想核心接近于人的思维，比较容易被用户接受，也是系统与用户之间最直接的沟通，所以概念模型的建立对后期设计合理的数据库来说是非常重要的。一般在数据库设计时，首先根据用户需求设计一个 E-R 图，然后将其转换成数据模型，再进行数据库设计。E-R 模型中主要有实体、联系和属性 3 类元素。

实体就是指现实世界中可以与其他事物相区别的事物。如为商场供应商品的供应商就是一个实体。每个实体都有一组属性，每个属性表示实体的一个方面，其中某个属性可以唯一地确定一个具体的实体。具有相同属性的实体的集合称为实体集。如供应商具有公司名称、联系人和地址等属性，一个商场中所有供应商的集合，可以称为供应商实体集。

联系分为实体内联系和实体集与实体集间的联系两种情况。实体内联系表示实体内各属性之间的联系，如员工的出生日期和员工的年龄之间就存在一种对应关系，将当前年份的值减去员工的出生年份，就可以得出员工的年龄。两个实体集间的联系又分为一对一、一对多和多对多 3 种联系类型。

- ❑ 两个实体集间的一对一联系：表示实体集 A 和实体集 B，实体集 A 中的一个实体最多只与实体集 B 中的一个实体相联系，记作 1：1。如学生实体集和班级实体集，学生实体集中的学生实体只与班级实体集中的一个班级相联系，所以称学生实体集与班级实体集是一对一的联系。
- ❑ 两个实体集间的一对多联系：表示实体集 A 和实体集 B，实体集 A 中的一个实体，与实体集 B 中的多个实体相联系，记作 1：n。如班级实体集和学生实体集，班级实体集中的一个班级实体，可以与学生实体集中多个学生实体相对应，所以称班级实体集与学生实体集是一对多的联系。
- ❑ 两个实体集间的多对多联系：表示实体集 A 和实体集 B，实体集 A 中的多个实体可以与实体集 B 中的多个实体相联系，记作 m：n。如学生实体集和课外活动小组实体集，学生实体集中的多名学生可以与课外活动小组实体集中的多个活动小组相联系，所以学生实体集和课外活动小组实体集之间是多对多的联系。

多个不同实体集之间的联系的情况比较复杂。以 3 个实体集为例，存在 1：1：1、1：1：n、1：m：n 和 r：m：n 共 4 种联系类型。

- ❑ 3 个实体集之间的 1：1：1 联系：表示实体集 A、实体集 B 和实体集 C，其中实体集 A 和实体集 B 之间是 1:1 联系的，实体集 A 和实体集 C 之间是 1：1 联系的，则称实体集 A、实体集 B 和实体集 C 之间是 1：1：1 联系的。
- ❑ 3 个实体集之间的 1：1：n 联系：表示实体集 A、实体集 B 和实体集 C，其中实体集 A 和实体集 B 之间是 1 对 1 的联系，而实体集 A 和实体集 C 之间是 1 对 n 的联系，则称实体集 A、实体集 B 和实体集 C 之间为 1：1：n 的联系。假设一个仓库只能由一个保管员负责，一个保管员也只能负责一个仓库，而一个仓库中可以存

放多种物品，一个物品也只能放在一个仓库中，此种情况下，仓库实体集、保管员实体集、物品实体集之间就是 1：1：n 的联系。

- 3 个实体集之间的 1：m：n 联系：表示实体集 A、实体集 B 和实体集 C，其中实体集 A 和实体集 B 之间是 1：m 的联系，而实体集 A 和实体集 C 之间是 1：n 的联系，则称其为 1：m：n 的联系。假设一个仓库可以由多个保管员负责，一个保管员只能负责一个仓库，而一个仓库中可以存放多种物品，此情况下，仓库实体集、保管员实体集和物品实体集之间就是 1：m：n 联系。
- 3 个实体集之间的 r：m：n 联系：表示实体集 A、实体集 B 和实体集 C，其中实体集 A 和实体集 B 之间是 r：m 联系，而实体集 A 和实体集 C 之间是 r：n 联系，则称实体集 A、实体集 B 和实体集 C 之间是 r：m：n 联系。如供应商、商场和商品的联系中，供应商为多个商场供应多种商品，每个商场可以出售多个供应商供应的多种商品，每种商品可以是多个供应商供应到不同商场中的商品。这时，称供应商、商场和商品是 r：m：n 联系。

同一实体集中也存在二元的联系，包括 1：1 联系、1：n 联系和 m：n 联系。

- 同一实体集中的 1：1 联系：如员工实体集中，婚姻联系就是 1：1 联系。
- 同一实体集中的 1：n 联系：如员工实体集中，领导和被领导联系就是 1：n 联系。
- 同一实体集中的 m：n 联系：如教职员工实体集中，学生和教师联系就是 m：n 联系。

实体集某方面的特性，就是实体集的属性。如供应商实体，属性有公司名称、联系人和地址等等。每个属性都有取值范围。如公司名称是长度不超过 100 个字符的字符串，联系人是长度不超过 20 个字符的字符串等等。同一实体集中每个实体的属性和属性的取值是相同的，但是，可能每个实体的取值并不相同。E-R 模型中的属性又有以下分类。

- 单值属性和多值属性：单值属性指实体集中只有一个取值的属性。如供应商实体的公司名称是单值属性。多值属性指实体集中可以有多个取值的属性。如员工信息中的员工工作经历，就是多值属性。
- 简单属性和复合属性：简单属性是指不可分割的属性。如供应商实体的公司名称就是一个简单属性。复合属性是指可以继续分割的属性。如供应商实体中的通信地址就是复合属性，可以继续细分为邮政编码、省、市和街道。
- 派生属性：派生属性是指从其他属性派生而来的属性。如员工实体中的年龄是由出生日期派生而来的。

11.3.2　E-R 设计

E-R 设计是一个复杂的过程，用户需要积累丰富的经验。以前面介绍过的供货商供货为例讲解进行 E-R 图设计的步骤，主要分为以下几步。

（1）抽象出需要的实体。如在本例中，主要是供货商实体、商品实体。

（2）抽象出系统中的关系实体。如在本例中，是供货商供货信息实体。

（3）抽象细化实体的属性。如在本例中，抽象出供货商的属性有供应商联系人和供应商地址，商品实体有商品单价属性。

（4）抽象细化关系实体的属性。如在本例中供货商供货除了供应商名称和商品名称外，还有供应的产品数量。

（5）确定各个实体的主键。如在本例中，供货商实体的主键是供货商公司名称（此处假定没有重名的供货商公司名称），产品实体的主键为供货的产品名称，供货商供货实体的主键为供货商公司名称和供货的产品名称的组合。

（6）确定各个实体之间的关系。如在本例中，供货商供货实体中的供货商公司名称与供货商实体的供货商公司名称相关联，供货的产品名称与产品实体的供货的产品名称相关联。

（7）确定需要建立的索引。如在本例中，因为要根据供货商地址查询，所以在供货商实体中在供货商地址上建立索引。

设计后的 E-R 图，如图 11-6 所示。

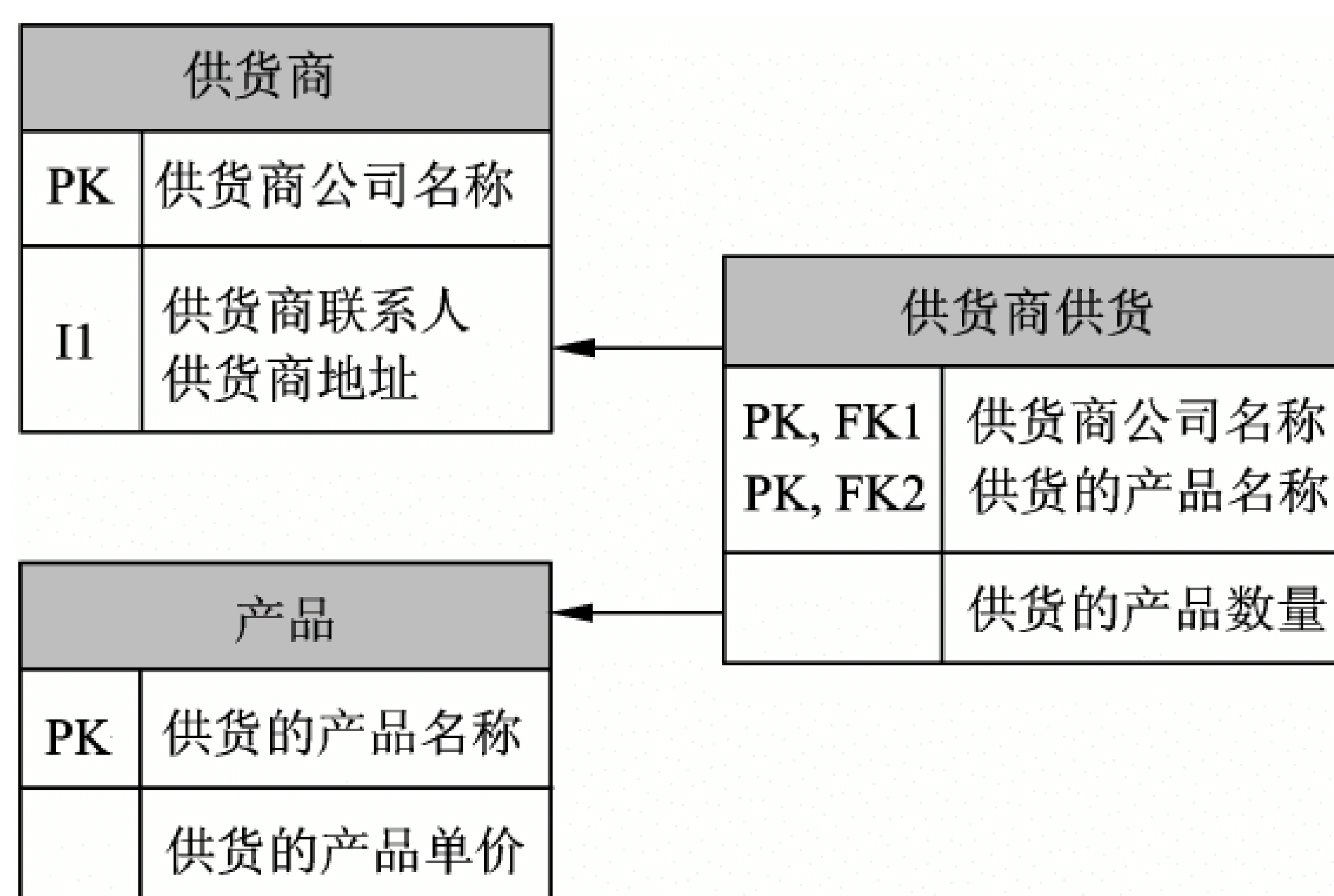


图 11-6　供货商供货 E-R 图示例

此处，仅是一个简单的 E-R 示例，读者需要根据自己的需求按照上面所讲的步骤详细地进行分析，抽象出自己的 E-R 图。

11.4　结构化查询语言 SQL

SQL 语言（Structured Query Language），即结构化查询语言，是查询、更新和管理关系型数据库的语言。作为业界的工业标准，目前市面上主流的数据库都提供了对 SQL 的支持，因此，对 SQL 的了解程度，直接影响着数据库程序的功能实现和效率。本节将介绍在 VC 中用到的 SQL 基本知识，有关 SQL 的详细知识，请参考有关 SQL 标准。

11.4.1　SQL 语言概述

SQL 是 1974 年由 Boyce 和 Chamberlin 提出的，1986 年成为关系数据库语言的美国标准，此后成为关系数据库语言的国际标准。因此，现在的各个数据库厂商基本上都支持 SQL 标准。当然在 SQL 基础上，根据自己数据库的特点做了一些扩展，所以各个数据库的 SQL 标准多少有些差异，但是基本上与 SQL 标准是一致的。所以读者在使用具体的数据库时，

可以参考 SQL 用户手册，注意各个数据库在对 SQL 的支持上的差异。SQL 是支持关系数据库的三级模式结构的，如图 11-7 所示。

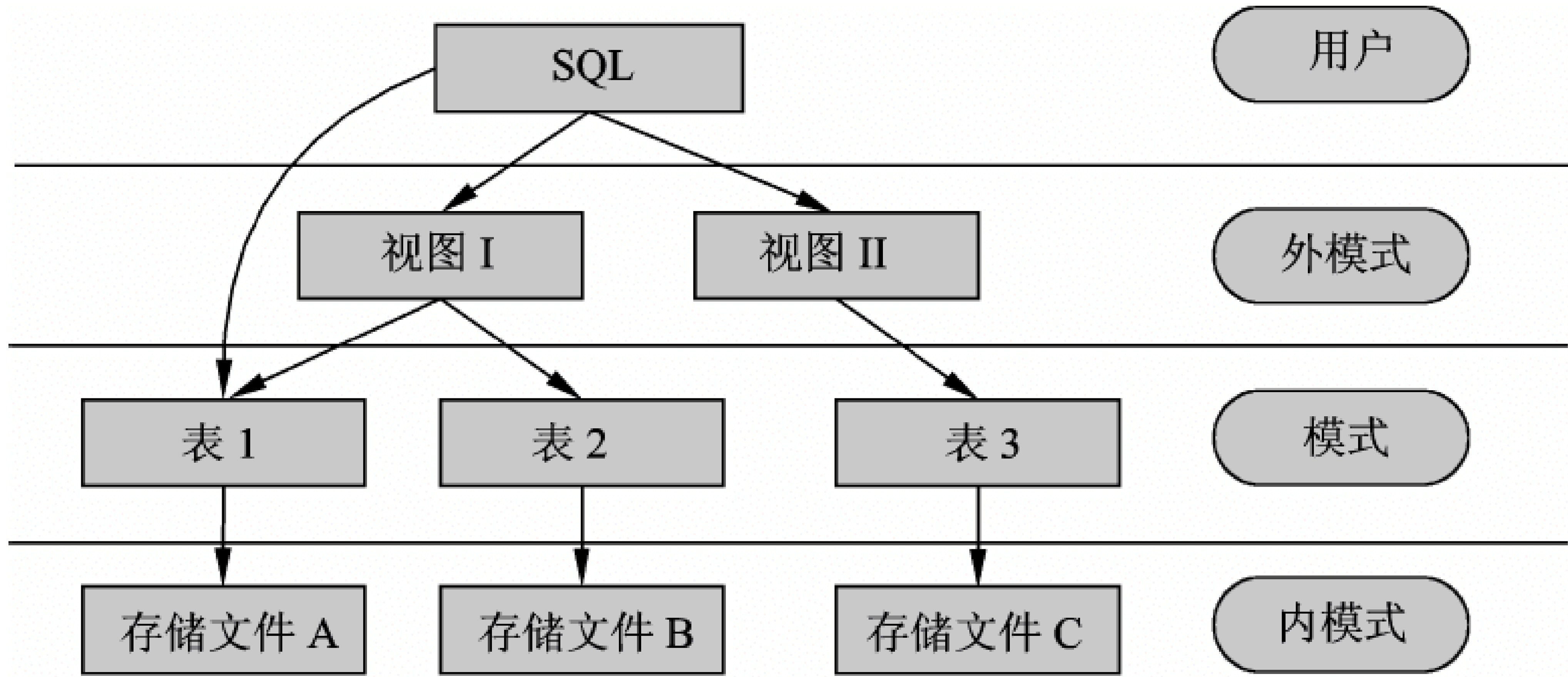


图 11-7　SQL 对关系数据库的三级模式的支持

在图 11-7 中，数据库存储文件 A、存储文件 B 和存储文件 C 都属于内模式，表 1、表 2 和表 3 属于模式，视图 I 和视图 II 属于外模式，对于用户开放的 SQL 属于用户接口层。

SQL 包括数据查询语言、数据操纵语言、数据定义语言和数据控制语言，是一个综合的、一体化的关系数据库语言。主要具有以下特点。

- 一体化。SQL 集数据库建立、查询、更新、维护和数据库安全性控制等功能为一体。
- 支持联机交互和嵌入语言两种使用方式，并且这两种方式的语法是统一的，为用户的使用和程序调试提供了方便。
- 语言简洁，简单易学。虽然 SQL 语言功能强大，但是其设计巧妙，围绕核心功能，规定了简单的语法，使用户可以快速入门。

由于目前大多数关系数据库都支持 SQL 标准，所以在进行数据库开发时，必须熟练掌握 SQL 语言。后面将详细讲述 SQL 语言的语法。前面介绍了数据库的基本概念，下面几小节将结合 SQL Server 中的 SQL 标准 ANSI SQL92 介绍关系数据库的语言——SQL 的基本知识。

11.4.2　SQL 数据定义语句 DDL

SQL 数据定义语句（Data Definition Language，DDL）用于定义和修改数据库对象。包括模式（表）、外模式（视图）和内模式（索引）。如用户可以使用 DDL 语言创建、修改和删除数据库的表、视图、触发器和存储过程等数据库对象。表 11-3 为 SQL 中的 DDL 语句。

表 11-3　SQL中的DDL语句

定 义 对 象	定 义 方 式		
	创　　建	删　　除	修　　改
表	CREATE TABLE	DROP TABLE	ALTER TABLE
视图	CREATE VIEW	DROP VIEW	
索引	CREATE INDEX	DROP INDEX	

从上表中可以看出，使用 SQL 定义语句可以定义表、定义视图和定义索引。但是，因为视图和索引是基于表之上的对象，所以，DDL 不提供视图和索引的修改语句，要修改视图和索引，可以先将其删除，然后重新创建。

DDL 使用 CREATE TABLE 语句创建表，其语法为：

```
CREATE TABLE <表名> (<列名><数据类型>[列的完整性约束条件]
                   [,<列名><数据类型>[列的完整性约束条件]…]
                   [,<表的完整性约束条件>])
                   [其他参数];
```

其中，表名表示要创建的表的名称。表定义列，除了需要为每列指定名称，即列名，还要为列定义数据类型和完整性约束条件。不同的 DBMS 支持的数据类型不完全相同，在定义时，应该支持使用的 DBMS 支持的数据类型。完整性约束条件会存入系统的数据字典中，当用户对表进行操作时，DBMS 会验证数据是否符合完整性约束条件，以保证数据的完整性。下面是在 SQL Server 2000 数据库中创建 Products 表的 SQL 语句。

```
CREATE TABLE Products (
    [ProductCode] [varchar] (15) COLLATE Chinese PRC CI AS NOT NULL ,
    [Description] [varchar] (50) COLLATE Chinese PRC CI AS NOT NULL ,
    [UnitOfMeasure] [varchar] (50) COLLATE Chinese PRC CI AS NOT NULL ,
    [UnitPrice] [decimal](10, 2) NOT NULL ,
    CONSTRAINT [PK Product] PRIMARY KEY  NONCLUSTERED
    (
    [ProductCode]
    )  ON [PRIMARY]
) ON [PRIMARY]
```

上面代码定义了 Products 表，共有 4 个字段，分别是 ProductCode、Description、UnitOfMeasure 和 UnitPrice，对应的数据类型分别是 varchar（15）、varchar（50）、varchar（50）和 decimal（10，2），这 4 个字段都不可以为 NULL，对于前 3 个字段使用中文的不区分大小写的、区别重音的中文排序规则，并且表使用 ProductCode 字段作为主键。

随着需求的变化，有时需要修改表的结构，此时，需要使用 ALTER TABLE 语句修改已经定义过的表，其语法格式为：

```
ALTER TABLE <表名>[ADD<新列名><数据类型>[列的完整性约束条件]]
                  [ADD <新列名><数据类型>[完整性约束条件]]
                  [DROP <完整性约束名>]
                  [MODIFY<列名><数据类型>];
```

其中，表名指定需要修改结构的表名，ADD 子句用于增加新列和新的完整性约束条件，DROP 子句用于删除定义的完整性约束条件，MODIFY 子句用于修改原来的列的定义。修改表不提供删除列的功能，如果要删除列，需要通过间接的方式实现。

对于不再需要的表，可以通过调用 DROP TABLE 语句删除。其语法格式为：

```
DROP TABLE <表名>;   // 表名指定了要删除的表的名称
```

为了提高查询效率，可以使用 CREATE INDEX 语句在表上定义索引，其语法格式为：

```
CREATE [UNIQUE][CLUSTER] INDEX <索引名>
                    ON <表名>(<列名>[<顺序>[,<列名>[顺序]]…]);
```

其中，索引名指定了创建索引的名称，表名指定了在其上创建索引的表的名称，列名指定了创建索引对应的列，顺序指定了该列的索引的顺序，UNIQUE 表示此索引对应的列的每个取值是否是唯一的，CLUSTER 表示索引项的顺序是否与表中记录的物理顺序一致。

索引的目的是提高查询效率，但是对于增删频繁的表来说，创建索引会降低效率。因此，有时根据需要，会遇到删除索引的情况。此时，可以使用 DROP INDEX 语句删除索引，其语法格式为：

```
DROP INDEX <索引名>; // 索引名指定了要删除的索引的名称
```

视图是外模式层次上的对象，是基于基本表而创建的，SQL 语句使用 CREATE VIEW 语句创建视图，其语法格式为：

```
CREATE VIEW<视图名> [(<列名>[,<列名>]…)]
                    AS <子查询>
                    [WITH CHECK OPTION]
```

其中，视图名指定了要创建的视图的名称。子查询可以是不包含 ORDER BY 子句和 DISTINCT 子句的任何 SELECT 子句。列名指定了视图的列，如果没有指定列，则视图会使用 SELECT 子句中的目标列作为列，但是当 SELECT 子句中的列为表达式或函数值，或者多表中有同名列或者在视图中要使用新列的情况下，则必须指定视图的列名。WITH CHECK OPTION 表示当对视图进行 INSERT、UPDATE 和 DELETE 操作时，检查操作的数据是否满足子查询中定义的条件。

当不再需要视图或者视图的基表被删除时，需要调用 DROP VIEW 语句删除视图，其语法格式为：

```
DROP VIEW<视图名>;   // 视图名指定要删除的视图的名称
```

11.4.3　SQL 数据操纵语句 DML

SQL 数据操纵语句（Data Manipulation Language，DML）用于完成数据查询和数据更新两种功能。数据查询功能包括查询语句（SELECT）。数据更新功能包括插入语句（INSERT）、删除语句（DELETE）和修改语句（UPDATE）。

数据查询是数据库最主要的操作之一，在 SQL 中使用 SELECT 语句实现查询功能，其语法格式为：

```
SELECT [ALL | DISTINCT] <目标列表达式>[,<目标列表达式>]…
            FROM <基本表或视图>[,<基本表或视图>]…
            [WHERE <条件表达式>]
            [GROUP BY <列名> [HAVING <内部函数表达式>]]
            [ORDER BY <列名> [ASC | DESC]];
```

其中，基本表或视图指定了要从其中选择数据的表或视图。目标列表达式指定了选择的目标列或表达式。条件表达式指定了查询数据使用的条件。GROUP BY 后的列名表示要进行分组的字段，HAVING 表示分组的依据。ORDER BY 后的列名表示要进行排序的字段，ASC 表示按照字段值升序排序，DESC 表示按照字段值降序排序。查询语句是 SQL 中最复杂的语句，根据目标列和条件表达式等部分的不同，还分为简单查询和嵌套查询和连接查

询等。这里就不详细讲述了。

数据库使用 INSERT 语句，可以将数据增加到数据库中，其语法格式为：

```
INSERT INTO 表名[(字段名[,字段名]…)]
                VALUES (常量[,常量]…);
INSERT INTO 表名[(字段名[,字段名]…)]
                子查询;
```

第一种格式表示，插入一条新记录，表名指定要插入数据的表的名称，字段名表示新插入记录的字段，常量表示要插入的记录对应的字段的取值；第二种格式表示，将查询到的记录集集体插入到表中，表名指定要插入数据的表的名称，字段名表示新插入记录的字段，子查询表示要插入的记录的选择插入条件。

对于表中不需要的记录，可以使用 DELETE 语句删除，其语法格式为：

```
DELETE FROM 表名
           [WHERE 条件]
```

其表示删除指定表中符合条件的记录。其中，表名指定要删除的记录所在的表。条件指定要删除的记录需要满足的条件，如果没有指定 WHERE 子句，则会删除表中所有的记录。DELETE 语句与 DROP 语句是不同的，DROP 语句是删除表结构，而 DELETE 语句是删除表中的记录。

在使用数据库中的数据时，可以根据需要使用 UPDATE 语句修改表中的记录，其语法格式为：

```
UPDATE <表名>
       SET <列名>=<表达式>[,<列名>=<表达式>]…
       [WHERE 条件]
```

其表示修改指定表中符合条件的记录的指定列的取值为指定的值。其中，表名指定要修改的数据所在表。条件指定要修改的记录需要满足的条件。列名表示要修改值的列的名称，表达式表示列修改后的取值。

11.4.4　SQL 数据控制语句 DCL

SQL 数据控制语句（Data Control Language，DCL）用于定义数据库的安全控制功能，其主要是对数据库中的对象的存取控制，即其规定不同的用户对不同的数据库对象具有不同的存取权限。如表 11-4 列出了数据库对象的操作种类。

表 11-4　数据库对象的操作权限类型

对象类型	对　　象	操作权限
TABLE	列	SELECT、INSERT、UPDATE、DELETE、ALL PRIVILEGES
	表	SELECT、INSERT、UPDATE、DELETE、ALTER、INDEX、ALL PRIVILEGES
	视图	SELECT、INSERT、UPDATE、DELETE、ALL PRIVILEGES
DATABASE	数据库	CREATE TABLE

其中，对于列和视图的操作权限有查询（SELECT）、插入（INSERT）、更新（UPDATE）和删除（DELETE）以及这 4 种权限的总和（ALL PRIVILEGES）。对于表的操作权限有查询（SELECT）、插入（INSERT）、更新（UPDATE）、删除（DELETE）、修改表（ALTER）和索引管理（INDEX）以及这 6 种权限的总和（ALL PRIVILEGES）。对于数据库的操作权限有创建表（CREATE TABLE）权限。

在数据库中，使用 DCL 的 GRANT 语句授予用户对指定对象的指定权限，其语法格式为：

```
GRANT <权限>[,<权限>…]
       [ON <对象类型><对象名>…]
       TO <用户>[,<用户>…]
       [WITH GRANT OPTION]
```

其中，对象类型和对象名指定了要授权权限的对象。用户指定了授权权限的用户，也可以使用 PUBLIC，将权限授予所有用户。权限指定要为用户授予的权限类型。而 WITH GRANT OPTION 表示，授权用户有权将这些权限授予其他用户。GRANT 语句可以一次为同一对象多个用户授权多个权限，但是不可以在一条 GRANT 语句中为两个对象类型授权，如不可以在一条 GRANT 语句中同时为 DATABASE 和 TABLE 两种对象授权。

数据库管理员可以根据需要使用 REVOKE 语句随时收回用户对指定对象的指定权限，其语法格式为：

```
REVOKE <权限>[,<权限>…]
        [ON <对象类型><对象名>…]
        FROM <用户>[,<用户>…]
```

其中，对象类型和对象名指定了要收回权限所在的对象，用户指定了要收回权限的用户，权限指定了要收回的权限类型。

11.4.5　操作视图

视图是基于基本表的数据库外模式层次上的对象，所以操作最终都会转换成对基本表的操作。合理的使用视图会带来下列优点。

- 视图可以简化用户查询数据的操作。使用视图，用户可以将其感兴趣的数据放到视图中，当用户需要对数据进行查询和操作时，只需要选择自己定义的视图就可以了，而不必在很多表中进行筛选，从而简化了用户的操作。
- 视图可以使用户从不同角度查询同一数据。因为同一数据对不同用户来说，意义是不同的，因此，不同用户可以根据自己的需求创建不同的视图，从而从不同的角度查看同一数据。
- 视图对保持数据的逻辑独立性提供了方法。因为视图对数据的呈现是基于用户的角度的，因此具有一定的逻辑独立性。当需要改变数据库模式层的设计时，可以尽可能地保持数据库的视图不变，从而减少开发工作量。
- 视图能够保持对机密数据的保密。可以根据需要，将用户需要的机密数据以视图

的方式公开，每个用户只能操作其需要的视图，而对于机密数据，只有具有权限的用户才有权进行操作。

有关视图的操作主要有查询视图、创建视图、删除视图和修改视图。它的操作语法与表的相应操作是相同的，有关视图的更新就是对基本表的更新。因为有些视图不能准确地对应到基本表上，因此，在数据库中有些视图是不可更新的。一般对行列子集的视图是更新的，其他情况是不可更新的。有关视图更新操作，各种 DBMS 的实现是不同的，因此，用户在使用视图更新操作时，应该确认对应的 DBMS 是否支持。

总之，SQL 语言在不同的 DBMS 上的具体实现细节不同，因此，用户在使用 DBMS 时，应该确认使用的 SQL 语言版本，根据实际情况使用 SQL 语句。

11.5 Visual C++数据库接口

虽然现在市面上存在多种 DBMS，每种 DBMS 的实现各不相同，使用的 SQL 语句版本不同，提供的数据管理工具也不相同，但是对于程序开发人员来说，在 Windows 平台下，微软提供了许多数据访问接口，可以抽象出数据访问的通用步骤，简化开发人员工作量。本节将介绍 Visual C++中可用的数据访问接口。

11.5.1 面向对象技术

前面讲过，现在最常用的是关系型数据库，而关系型数据库的核心是将概念模型中的实体和关系都抽象成实体。这就要用到有关面向对象的技术。所谓面向对象，就是将机器世界中处理的数据都看作对象处理，将数据的特点作为对象的属性，将对数据的操作作为对象的方法。

使用面向对象技术进行编程，思维模式更接近于现实世界。因此，在 Windows 平台下，将面向对象的技术集成到数据访问技术中。因为在第 4 章中，曾详细讲述过有关面向对象的技术，因此这里就不再重复了。

11.5.2 Windows 平台下的数据访问接口

微软在 Windows 平台下提出了跨企业的信息访问策略，即通用数据访问。提供高性能的访问各种关系型信息源和非关系型信息源，并提供独立于开发工具和开发语言的易于使用的编程接口。使用这些技术，可以集成多种数据源，创建易于维护的解决方案，并且可以使用选择的最好的工具、应用程序和平台服务。

通用数据访问不需要将不同 DBMS 中的数据存储到单个数据库中，也不需要将数据库局限于单个数据库提供商。通用数据访问是具有广泛工业支持的开放的工业标准，支持所有主流的数据库平台。图 11-8 中列出了微软提供的通用数据访问中包含的技术。

从图 11-8 中可以看出，在 Windows 平台下，提供了多种访问不同数据源的接口。包

括以下几部分。

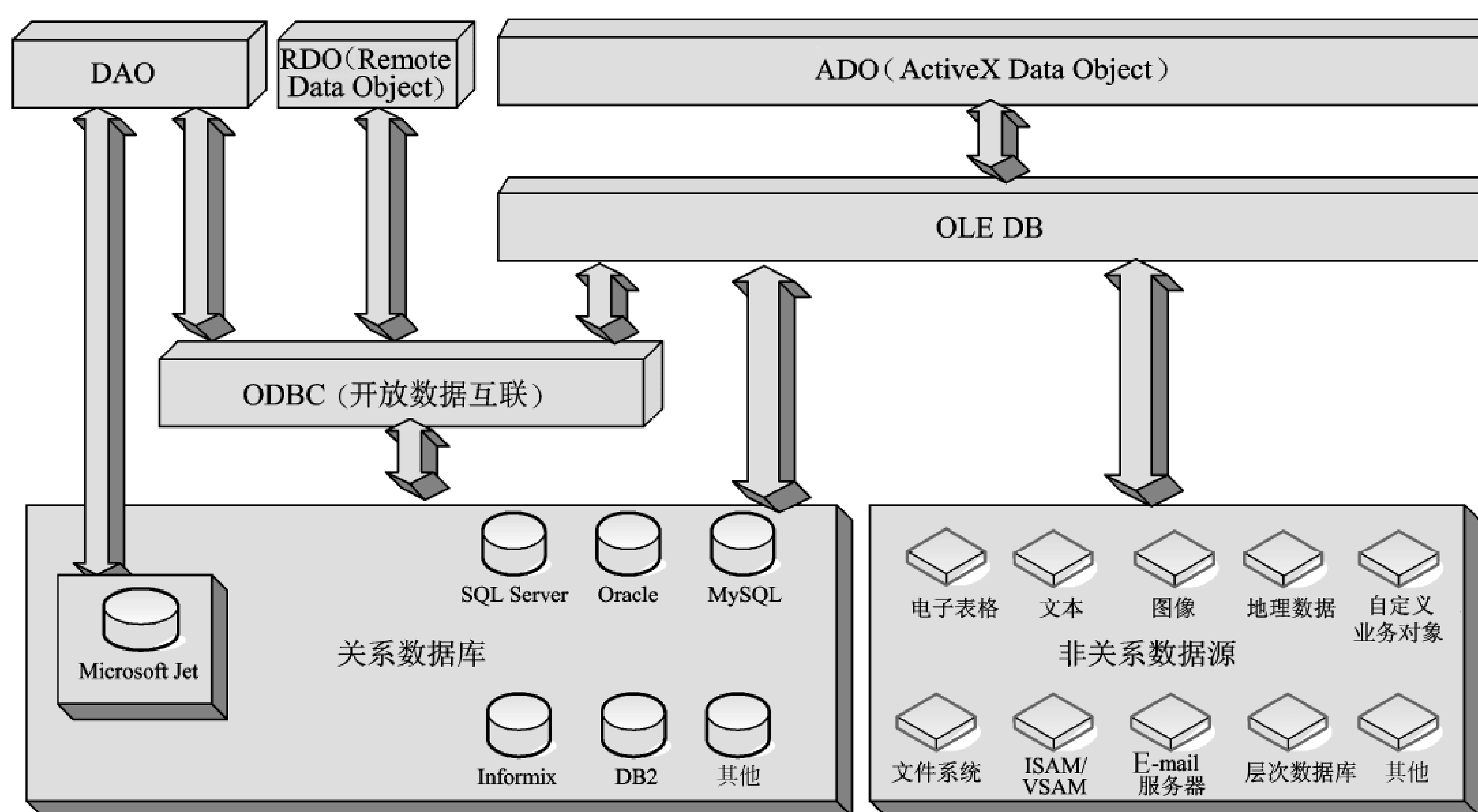


图 11-8　微软通用数据访问接口

ODBC（Microsoft Open Database Connectivity，开放数据互联）是一组工业标准，并且是 Windows 开放服务架构 WOSA（Microsoft Windows Open Services Architecture）的一个组件。ODBC 接口提供到各种不同的关系型数据库的接口，提供了极大的互操作性，即应用程序可以通过单个接口从数据源中访问数据。这使得应用程序可以独立于 DBMS 访问数据。如目前市场上主流的数据库，SQL Server、Oracle、MySQL、Informix 和 DB2 等都可以通过 ODBC 访问。而且用户可以根据需要，增加相应数据源的驱动，扩展 ODBC 所支持的数据源。ODBC 通过驱动提供应用程序和 DBMS 之间的接口。

OLE DB 是跨组织的系统级数据编程接口，是访问所有数据种类的公开标准。ODBC 用于访问关系数据库，而 OLE DB 设计用于访问包括关系型和非关系型数据源在内的所有数据源种类，包括大型机 ISAM/VSAM、层次数据库、E-mail 和文件系统、文本、图像、地理数据、自定义业务对象等。ODBC 定义了一组 COM 接口，这些接口完成对数据源的访问。OLE DB 包括数据提供组件、用户组件和服务组件。其中，数据提供组件公开数据；用户组件用于使用数据；服务组件用于处理和传输数据。OLE DB 接口可以实现平滑的集成。因此，数据提供厂商可以通过 OLE DB 组件为用户提供稳定高效的数据访问组件。另外，从图 11-8 可以看出，OLE DB 还提供了对 ODBC 的集成，使用户使用 OLE DB 也可以通过 ODBC 访问主流的关系数据库，当然，用户可以通过各关系型数据库的 OLE DB 驱动直接访问数据。

ADO（Microsoft ActiveX Data Objects，ActiveX 数据对象）是对 OLE DB 接口的封装，以 COM 组件的形式供用户使用，提供统一的高效数据访问接口，支持多种开发环境。既可以创建前台数据库客户端，也可以创建 Web 浏览器使用的中间业务对象。同时，可以在主流的开发工具、数据库工具、开发语言和开发工具上使用。优点是使用方便、速度快、

占用内存小。

DAO（Data Access Object，数据访问对象）是微软特意为 Microsoft Jet 引擎数据库而设计的数据访问对象，对 Microsoft Jet 引擎的数据访问是非常高效的，同时，还提供对 ODBC 的访问。

RDO（Remote Data Object，远程数据对象）是对 ODBC 接口的封装，以 COM 组件的形式供用户使用，主要是为远程数据访问而设计的。随着其他数据访问技术的发展，现在不提倡使用 RDO。

其中，ADO、OLE DB 和 ODBC 技术组成了数据访问组件（MDAC，Microsoft Data Access Components），是微软通用数据访问技术的核心，具体每项技术的使用在后面的章节中会详细介绍。

11.5.3　Visual C++数据访问接口

为了简化编程过程，Visual C++在数据访问接口之上提供对其的封装，如图 11-9 所示。

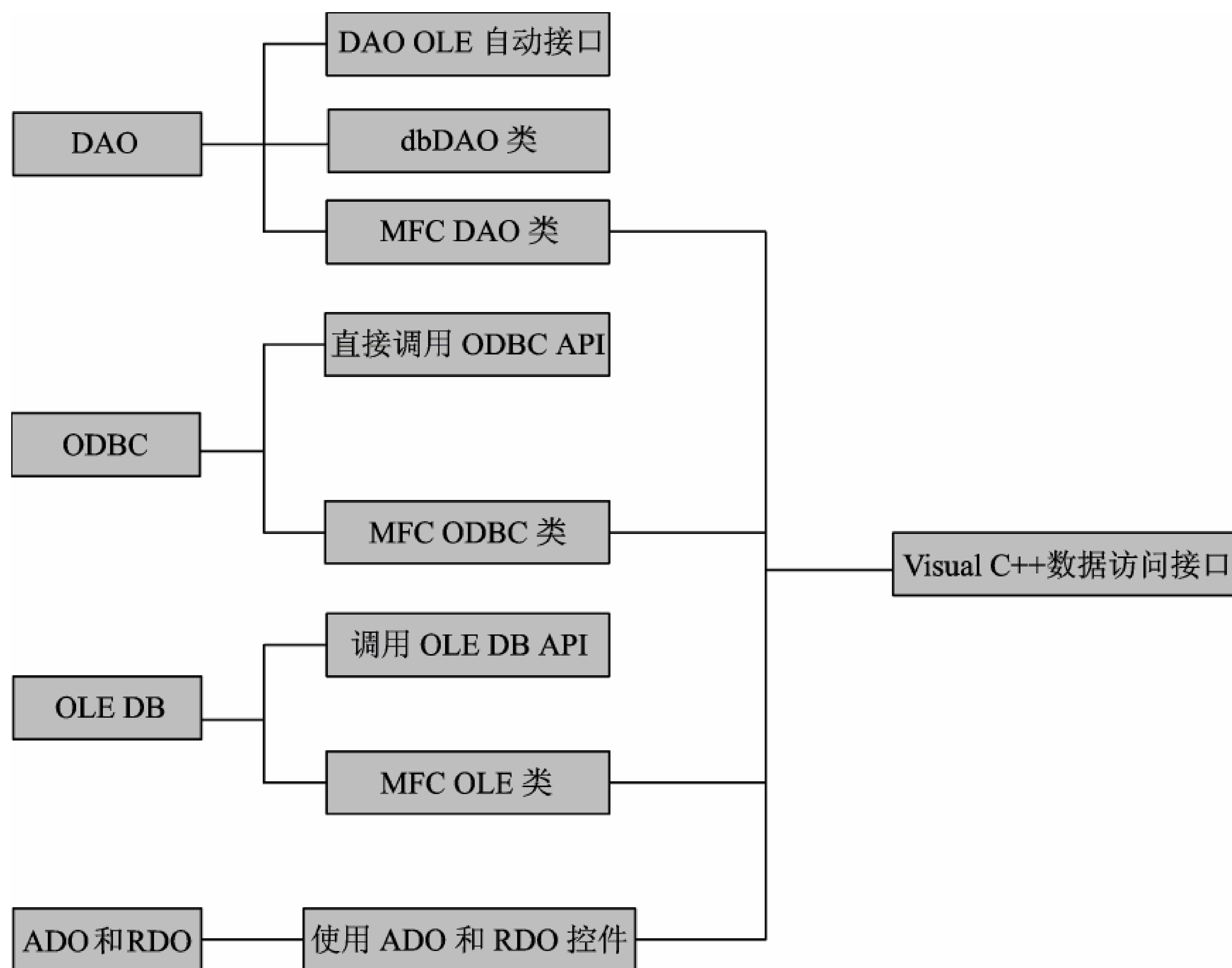


图 11-9　Visual C++的数据访问接口

从图 11-9 中可以看出，在 VC 中有 3 种方式可以使用 DAO 进行编程。第一种是直接使用 DAO OLE 自动接口，因为 DAO 是一组 OLE 接口，因此用户可以像使用其他 OLE 组件一样，使用 DAO OLE 自动接口，这种方式的优点是可以充分使用 DAO 的所有功能，但是，开发过程复杂繁琐，因此，一般不使用这种方式；第二种是 dbDAO 类，是对 DAO 的封装，目的是为了简化调用 DAO OLE 自动接口的复杂性；第三种是 MFC DAO 类，是 MFC 对 DAO 的封装，将 DAO 中常用的数据库操作封装成 MFC 类，使得 DAO 的开发非

常简便，开发人员不需要了解具体的 DAO 接口，而只需要了解对应的 MFC 类即可，从而简化了开发人员的工作量。

在 VC 中使用 ODBC 进行开发有两种方式，一种是直接调用 ODBC API，用户可以通过这种方式充分使用 ODBC 的所有功能，如可以通过 ODBC 调用 DDL 语句；另一种是 MFC ODBC 类，是 MFC 对 ODBC 的封装，将 ODBC 中常用的数据库操作封装成 MFC 类，使得 ODBC 的开发非常简便，开发人员不需要了解具体的 ODBC 接口，而只需要了解对应的 MFC 类即可，从而简化了开发人员的工作量，类模型结构与 MFC DAO 非常类似。但是使用 MFC ODBC 类也有局限性，如使用 MFC ODBC 类不可以在程序中调用 DDL 语句，如果要调用 DDL 语句，则必须直接调用 ODBC API。

同样在 VC 中使用 OLE DB 进行开发也分两种方式，一种是直接调用 OLE DB API，此种方式的特点是功能齐全，但是编程复杂繁琐，所以，一般不常用；另一种是常用的 MFC OLE 类，MFC 为 OLE DB 的调用提供了一组常用类 CDataSource、CSession、CRowSet 和 CTable 等，使用这些类就可以直接通过 OLE DB 访问数据。

ADO 和 RDO 在 VC 中的使用比较方便，就像使用普通的 ActiveX 控件一样。具体的操作步骤会在第 12 章中介绍。

11.5.4　用 Visual C++访问数据库的优点

在 Visual C++中访问数据库，具有很多优点。

- 简单性：VC 为数据访问提供了很多自定义的接口，这些接口简单、易用，简化了程序开发的工作量。
- 灵活性：VC 根据数据源的不同，为数据访问提供了多种接口，用户可以根据自己的需要选择合适的接口。如对于 Microsoft Jet 引擎的数据源，选择 DAO 比较好。对于非关系型数据源，则应该选择 OLE DB 数据接口。而对于数据库客户端程序，则使用 ADO 开发，比较合适。
- 速度快：实际上，这也是 VC 的一个特点，处理速度比其他语言要快，使用它访问数据库，运行速度比较快。
- 可扩展性：因为 VC 提供的这些数据访问接口，大部分是独立于不同数据源的，因此用户可以抛开底层数据源的限制，编写统一的数据库应用程序，即使当数据源发生变化时，也不需要改动应用程序。如 ODBC 就是对所有的关系型数据库的抽象，而 OLE DB 则是对所有数据源的继承，使用户可以根据需要扩展应用程序的数据源。

总之，使用 VC 访问数据库是简便、高效、可扩展的方式。

11.6　本 章 小 结

本章介绍了数据库的基本知识，包括数据库简介、规范化理论、E-R 模型、结构化查

询语言 SQL 和 VC 提供的数据库接口。本章重点是掌握数据库概念并理解 VC 数据库接口，为学习后面几章数据库访问打好基础。第 12～16 章将分别介绍各种数据库访问技术，第 12 章首先介绍在 VC 中访问 SQL Server 的技术。

11.7　习　　题

1．解释关系数据库中字段、记录和主键的概念。

【思路】关系数据库的相关概念在 11.1.7 小节中有描述，可以理解了以后再用自己的语言描述出来。

2．参考 11.3.2 小节的供货商供货 E-R 图示例，作出一个班级、学生、课程和教师 4 者关系的 E-R 图。

【思路】先深刻理解 11.3 节所讲的内容，然后作图。

第 12 章 Visual C++中 SQL Server 访问技术

第 11 章介绍了有关数据库的理论知识，从本章开始，将结合 Visual C++的数据库访问技术介绍有关数据库的实际操作知识。本章先以 SQL Server 2008 为例，介绍在 Visual C++中使用 ADO 进行数据库编程的步骤和注意事项。

12.1 SQL Server 2008 简介

SQL Server 最初是由 Microsoft、Sybase 和 Ashton-Tate 三家公司共同开发的关系数据库管理系统。在 Windows NT 推出后，Microsoft 与 Sybase 分别专注于 SQL Server 在 NT 和 Unix 上的开发。经过近 10 年的发展，SQL Server 在性能和可靠性上都在不断地提高。本章将着重介绍 SQL Server 2008 的特性和使用。

12.1.1 SQL Server 2008 介绍

SQL Server 2008 是一款全面的数据库管理系统。通过集成的商业智能工具，它提供了企业级的数据管理，提供了以下组件为用户提供数据服务。

- ❑ Microsoft SQL Server 2008 Database Engine：用于存储、处理和保护数据的核心服务。利用数据库引擎可控制访问权限，并快速处理事务，从而满足企业内要求极高而且需要处理大量数据的应用需要。数据库引擎为保持高可用性方面提供了有力的支持。
- ❑ Microsoft SQL Server 2008 Analysis Services（SSAS）：为商业智能应用程序提供了联机分析处理（OLAP）和数据挖掘功能。通过 Analysis Services 可以设计、创建和管理包含多维结构的数据结构，包含从其他数据源（如关系数据库）聚合的数据，并通过这种方式支持 OLAP。对于数据挖掘应用程序，通过 Analysis Services 可以实现多种行业标准的数据挖掘算法设计、创建和可视化，并能实现从其他数据源构造的数据挖掘模型。
- ❑ Microsoft SQL Server 2008 Integration Services（SSIS）：是生成高性能数据集成解决方案的平台，包括数据仓库的提取、转换和加载。Integration Services 包含用于生成和调试包的图形工具及向导；用于执行工作流功能的任务，如 FTP 操作、SQL 语句执行和电子邮件消息处理；用于提取和加载数据的数据源和目标；用于清理、

聚合、合并和复制数据的转换；用于管理 Integration Services 的管理服务 Integration Services；以及对 Integration Services 对象模型进行编程的应用程序编程接口。

- 复制：是在数据库之间对数据和数据库对象进行复制和分发，然后在数据库之间进行同步以保持一致性的一组技术。使用复制可以将数据通过局域网、广域网、拨号连接、无线连接和 Internet 分发到不同位置以及分发给远程用户或移动用户。
- Microsoft SQL Server 2008 Reporting Services：是一种基于服务器的解决方案，用于生成从多种关系数据源和多维数据源提取内容的企业报表，发布能以各种格式查看的报表。Reporting Services 包含用于创建和发布报表及报表模型的图形工具和向导；用于管理 Reporting Services 的报表服务器管理工具；以及用于对 Reporting Services 对象模型进行编程和扩展的应用程序编程接口。
- Microsoft SQL Server 2008 Notification Services 平台：用于开发和部署可生成并发送通知的应用程序。可以使用 Notification Services 生成并向大量订阅方及时发送个性化的消息，还可以向各种各样的设备传递消息。
- Microsoft SQL Server 2008 引入了 Service Broker，这是一项全新的技术，可用于生成数据库加强型的安全、可靠、可扩展的分布式应用程序。
- Microsoft SQL Server 2008 包含对 SQL Server 表中基于纯字符的数据进行全文查询所需的功能。全文查询可以包括单词和短语，或者一个单词或短语的多种形式。
- Microsoft SQL Server 2008 提供了设计、开发、部署和管理关系数据库、Analysis Services 多维数据集、数据转换包、复制拓扑、报表服务器和通知服务器所需的工具。

12.1.2　SQL Server 2008 的工具

Microsoft SQL Server 2008 包括一组完整的图形工具和命令行实用工具，这有助于用户、程序员和管理员提高工作效率。SQL Server 2008 包括的实用工具有以下几个。

- SQL Server Management Studio：是 SQL Server 的集成开发环境，可以用于访问、配置、管理和开发 SQL Server 项目。
- Business Intelligence Development Studio：用于生成对象和构造的工具，主要用于实现数据提取、数据转换、数据分析和数据报告的功能。
- SQL Server Profiler：用于监视 SQL Server Database Engine 或 Analysis Server 实例的 SQL 跟踪图形工具。
- SQL Server 配置管理器：用于启动、停止和配置与 SQL Server 相关的服务的工具。
- 命令提示实用工具：用于移动大容量数据、运行脚本以及管理 Notification Services 等操作的命令提示工具。
- SQL Server 2008 用户界面参考：提供各种工具的帮助文件。
- SQL Server 工具教程：有关如何使用 SQL Server Management Studio、sqlcmd 实用工具和数据库引擎优化顾问的指导课程。

12.1.3　SQL Server 2008 配置管理器

使用 SQL Server 2008 首先需要启动 SQL Server 2008 相关的服务，这些服务也就是

SQL Server 2008 的服务器组件。只有启动了服务器组件，用户才可以通过应用程序或客户端工具访问 SQL Server 2008 数据库，执行数据库操作。

虽然用户可以在 Windows 服务中操作这些服务，但是在 Windows 服务中进行这些操作比较繁琐，因此 SQL Server 2008 为用户提供了操作服务的图形化实用工具——SQL Server 配置管理器。配置管理器可以实现对 SQL Server 2008 的服务的启动、暂停和停止操作。使用方法是选择“开始”|“所有程序”|Microsoft SQL Server 2008|“配置工具”|SQL Server Configuration Manager 命令，打开 SQL Server Configuration Manager 对话框，如图 12-1 所示。

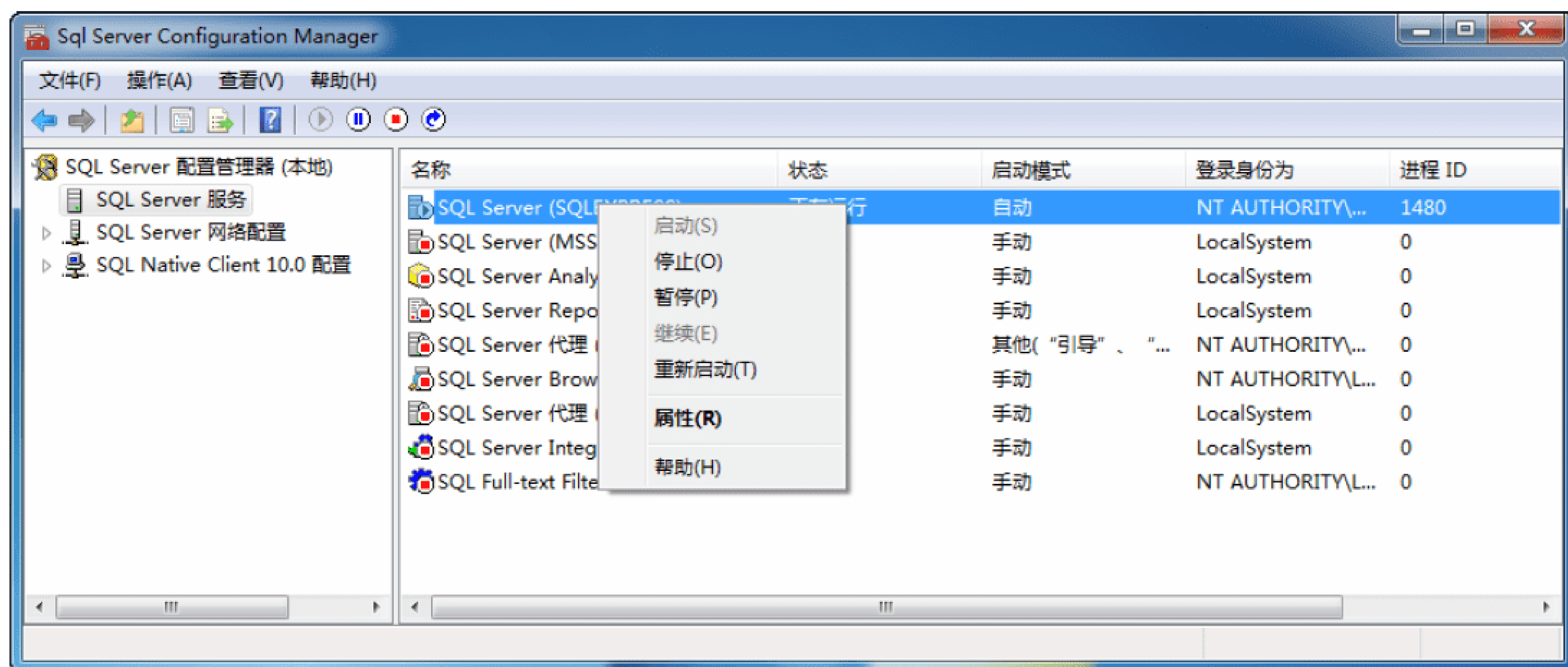


图 12-1　SQL Server 服务管理器工作界面

其中，图 12-1 中左边的树形视图是配置管理器可以管理的服务项，右边的列表视图是选择的服务对应的服务组件的状态。如果想要操作指定的服务项，只需右击相应的服务，在弹出的快捷菜单中选择相应的命令即可。例如在图 12-1 中，右击 SQL Server 服务，在弹出的快捷菜单上，如果选择“停止”命令，则可以停止 SQL Server 服务。其他操作依此类推。

12.1.4　SQL Server Management Studio

SQL Server 2008 为用户提供了一个管理数据库的图形化工具——Microsoft SQL Server Management Studio。使用方法是选择“开始”|“所有程序”|Microsoft SQL Server 2008|SQL Server Management Studio 命令，打开 Microsoft SQL Server Management Studio 对话框，如图 12-2 所示。

如图 12-2 所示，整个 SQL Server Management Studio 主要包含以下几部分。

- 标题栏：显示当前选择的对象的信息，它随着用户选择对象的变化而变化。如当用户选择数据库时，标题栏上显示相应数据库路径及其名称。
- 菜单栏：显示常用的命令菜单。由于 SQL Server Management Studio 的框架使用的是 Windows 的控制台管理器，所以根据对象的不同，特性化的菜单分别在“操作”菜单下和“工具”菜单下，这里就不再详细讲述了，在 12.2.1 小节中会讲述其部分功能的使用。

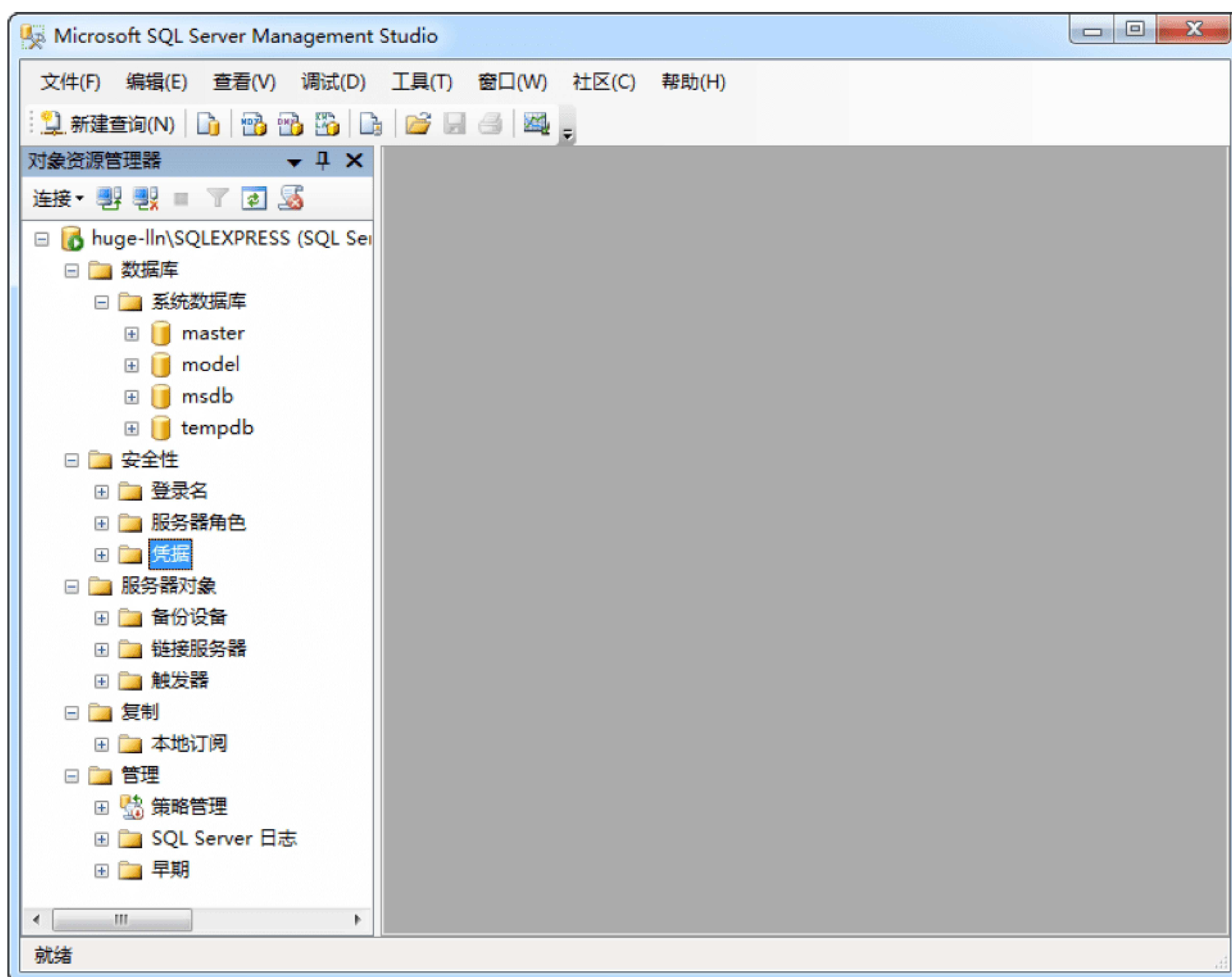


图 12-2　SQL Server Management Studio 工作界面

- 工具栏：显示了常用的命令快捷键，也就是部分菜单项。工具栏的设置是为了方便用户的操作。
- 对象资源管理器：其中显示了 SQL Server 2008 的服务实例对象。每个实例包括数据库、对应的安全性设置、服务器对象、复制设置和管理等其他数据选项组。用户主要是通过单击对象资源管理器中相应的对象进行具体操作。
- 工作区：用户选择要操作的对象后，在工作区中执行具体的操作。如查询数据、创建数据库对象等操作。这个区域是用户的工作区域。
- 状态栏：状态栏中显示当前操作的状态。

SQL Server 2008 Management Studio 的工作界面主要就是这几部分。使用 SQL Server 2008 Management Studio 可以完成 SQL Server 2008 的大部分数据操作，所以读者应该熟练掌握其操作方法。

12.2　创建 SQL Server 2008 对象

Microsoft SQL Server 2008 Management Studio 提供了可视化的创建数据库对象的方法，支持 SQL Server 2008 的对象有数据库、表、视图、存储过程、触发器、函数和用户自定义数据类型等。本节简单地介绍使用 Microsoft SQL Server 2008 Management Studio 如何创建这些 SQL Server 2008 对象。

12.2.1　创建用户数据库

数据库是管理数据的单位，其中可以包含表、视图、存储过程、触发器和函数等 SQL Server 2008 对象。在 SQL Server 2008 Management Studio 中创建数据库的步骤如下。

（1）打开 SQL Server 2008 Management Studio，展开 SQL Server 组到数据库层。在“数据库”文件夹下右击，如图 12-3 所示。

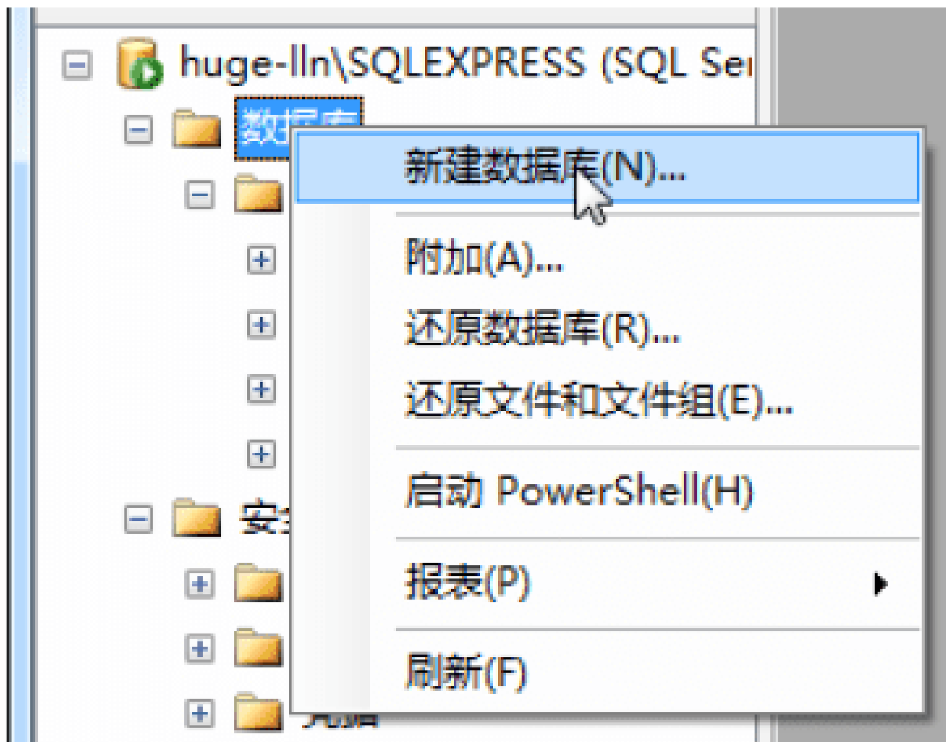


图 12-3　创建数据库——第一步

（2）选择弹出菜单中的“新建数据库”命令，打开“新建数据库”对话框，如图 12-4 所示。

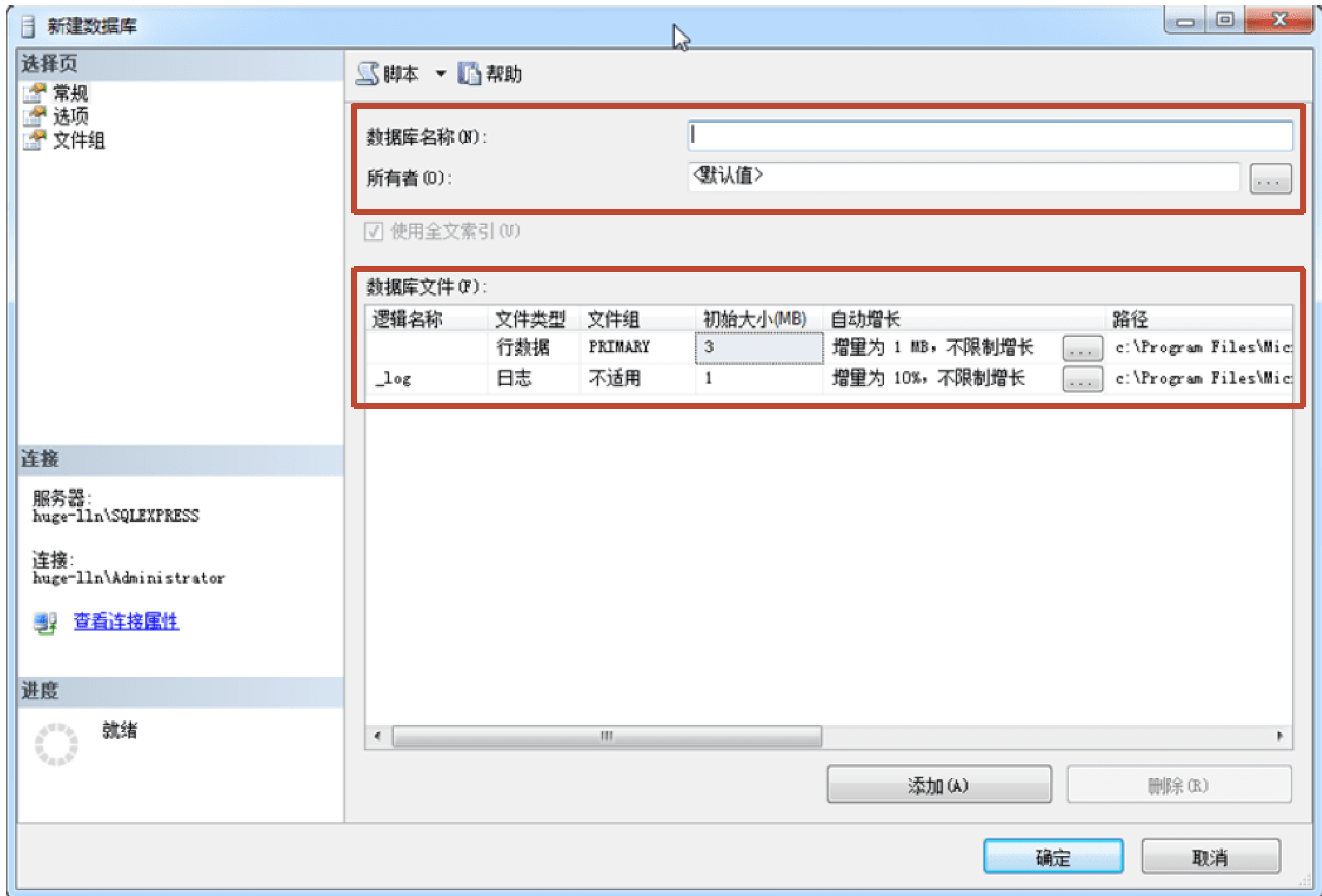


图 12-4　创建数据库——第二步

在“数据库名称”文本框中输入要创建的数据库的名称，在“数据库文件”列表中设置数据文件和日志文件的名称、初始大小、自动增长和路径属性。单击“确定”按钮，用

户数据库就创建成功了。

12.2.2　创建和管理表

数据表是数据库中最基本的元素，其中存储着最基本的原始数据。在 SQL Server 2008 Management Studio 中创建表的步骤如下。

（1）打开 SQL Server 2008 Management Studio 中，展开 SQL Server 组到要创建表的数据库层，如图 12-5 所示。

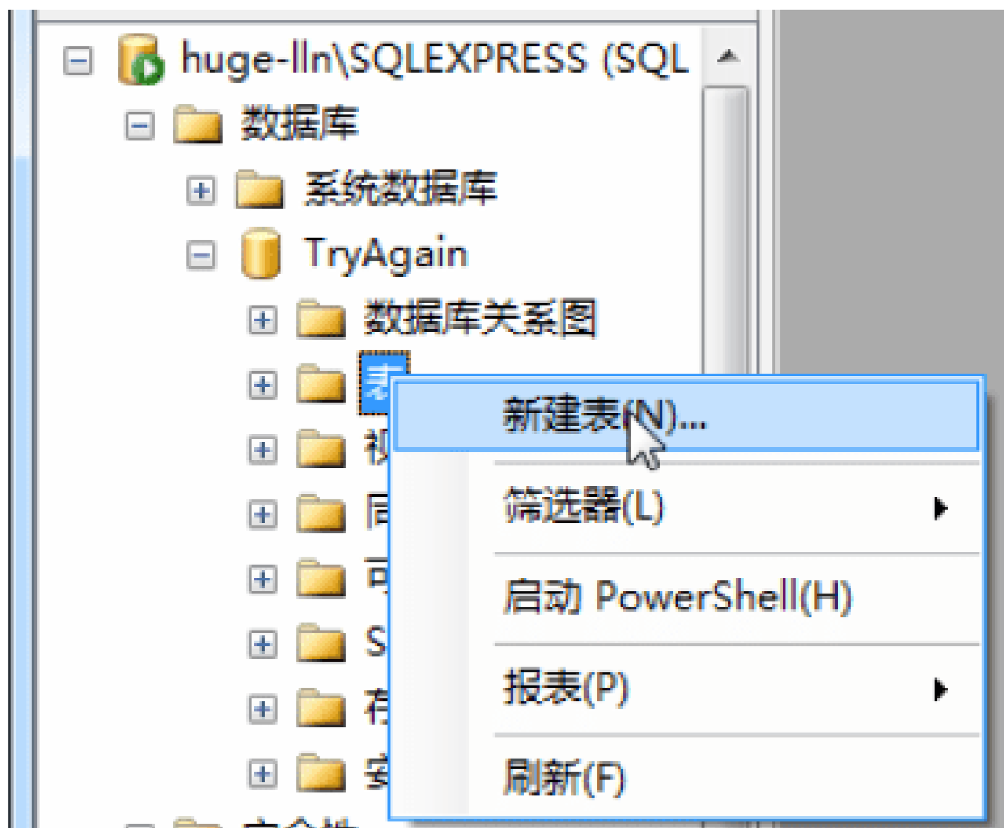


图 12-5　创建表——第一步

（2）选择弹出菜单中的“新建表”命令，打开新表面板，如图 12-6 所示。

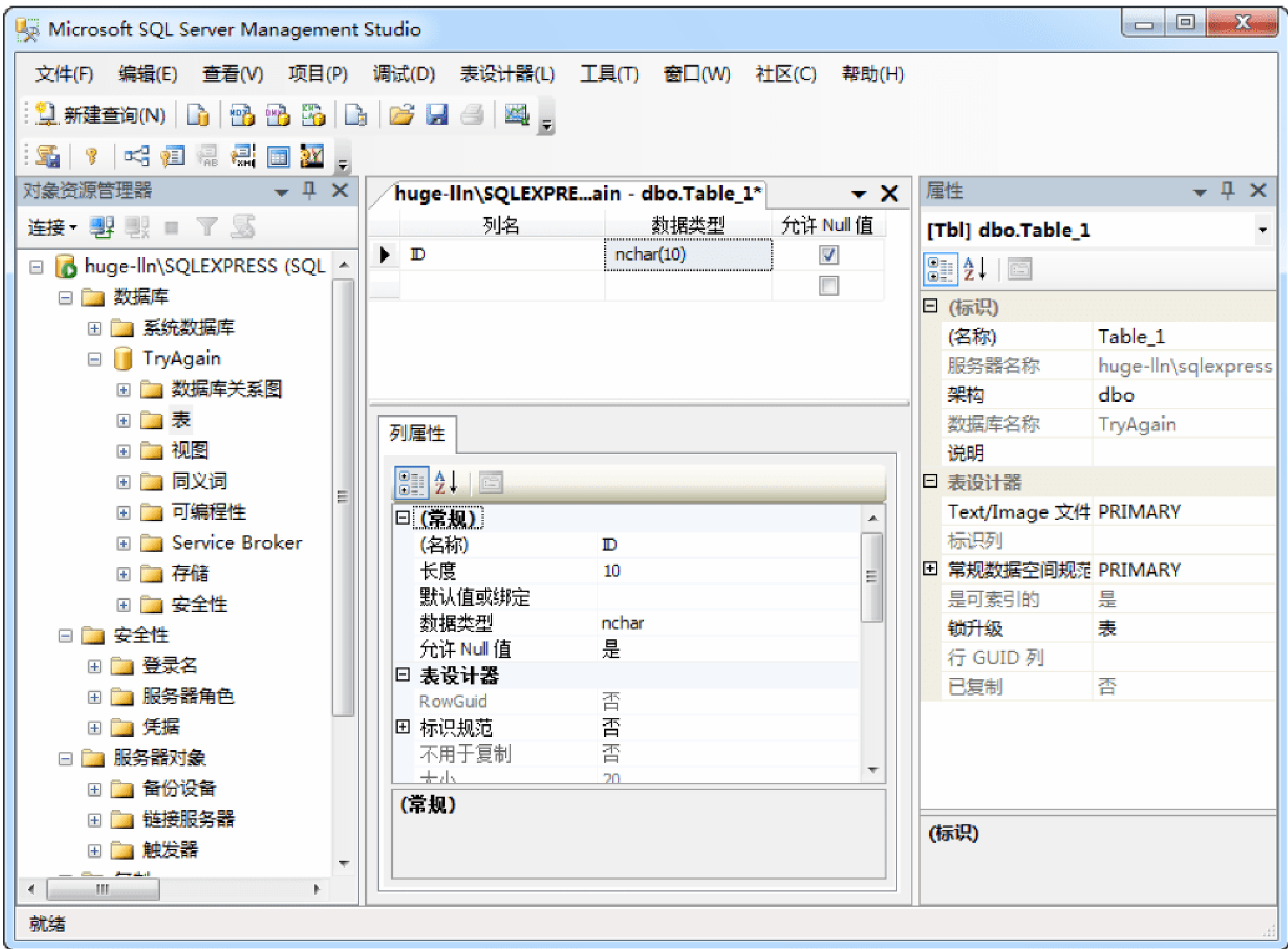


图 12-6　创建表——第二步

在图 12-6 中，在右边“属性”标签中的“名称”文本框中输入表名。在列表框中的“列

名”列中输入要创建的表的字段名称，“数据类型”列中指定要创建的表的字段的类型，“允许空”列中指定要创建的表的字段是否允许设置为空值。在下面的“列”标签中的“说明”文本框中输入此字段中存储的数据的含义，在“默认值”文本框中输入字段的默认值，在“精度”文本框中输入数字型字段的精度，在“小数位数”文本框中输入数字型字段的数据的小数位数，在“标识”列表框中选择字段是否是标识字段，如果是标识字段，则在“标识种子”文本框中输入标识字段的种子，即开始的计数值，在“标识增量”文本框中输入标识字段每次递增的数量。在“排序规则”文本框中选择对字段排序的规则。

（3）输入完这些信息，单击“保存”按钮或退出，用户数据表就创建成功了。

（4）当需要重新修改表设计时，右击表名，在弹出的快捷菜单中，单击“修改”命令，按照新建表的过程修改表设计后，单击“保存”按钮或“退出”按钮即可。

12.2.3　创建和管理视图

视图是一种将数据库基本表中的数据进行组合，方便查看的数据库对象。在 SQL Server 2008 Management Studio 中创建视图的步骤如下。

（1）打开 SQL Server 2008 Management Studio 中，展开 SQL Server 组到要创建视图的数据库层，在“视图”结点上右击。

（2）选择弹出菜单中的“新建视图”命令，打开“新视图”对话框，如图 12-7 所示。

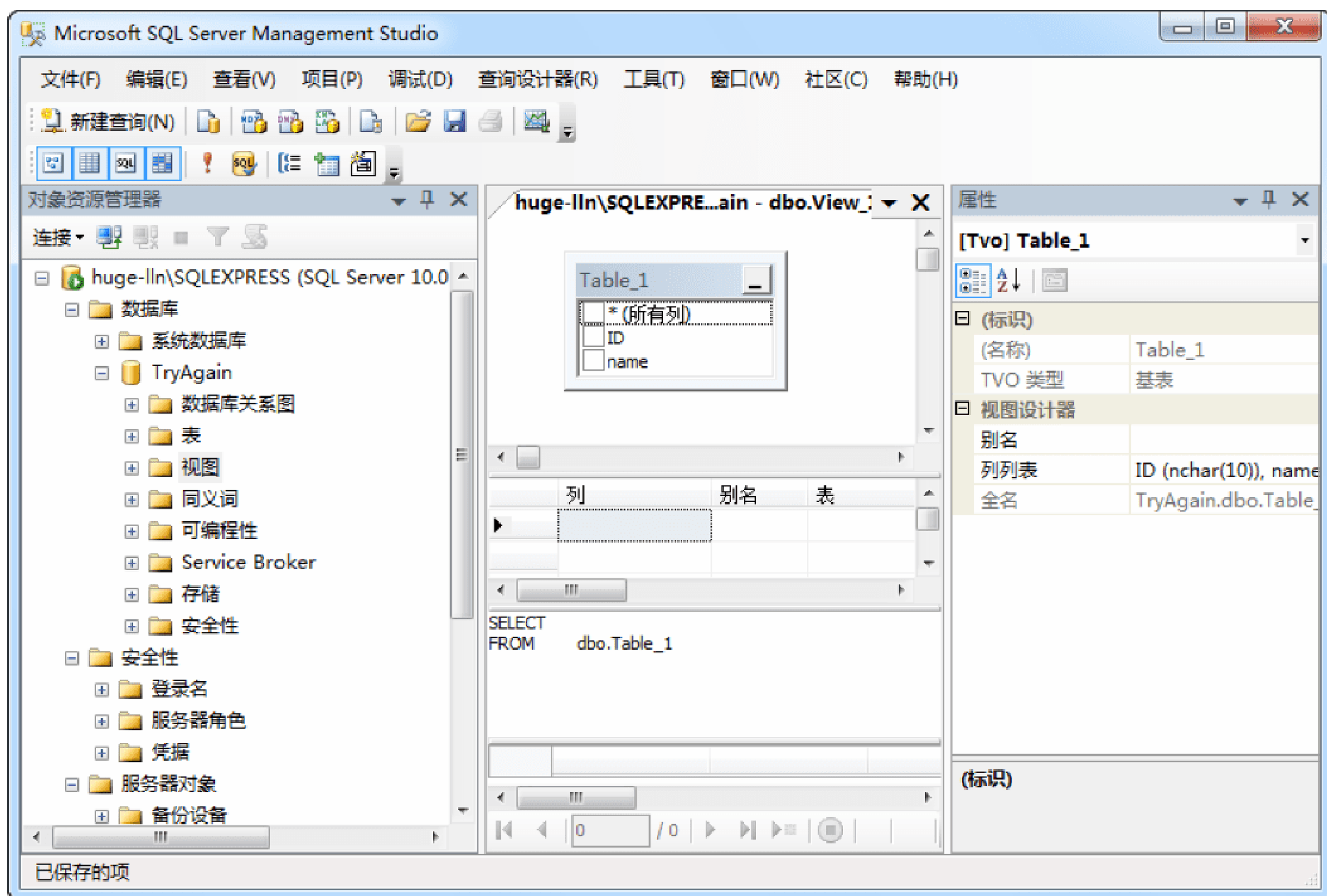


图 12-7　创建视图

在中间部分的文本框中输入视图中要选择的数据的 SQL 语句，或是通过可视化的界面添加要选择的表和字段。

（3）输入完这些信息，单击工具栏中的保存按钮，打开“选择名称”对话框，如图 12-8 所示。在“输入视图名称”文本框中输入要保存的视图的名称，单击“确定”按钮，用户数据视图就创建成功了。

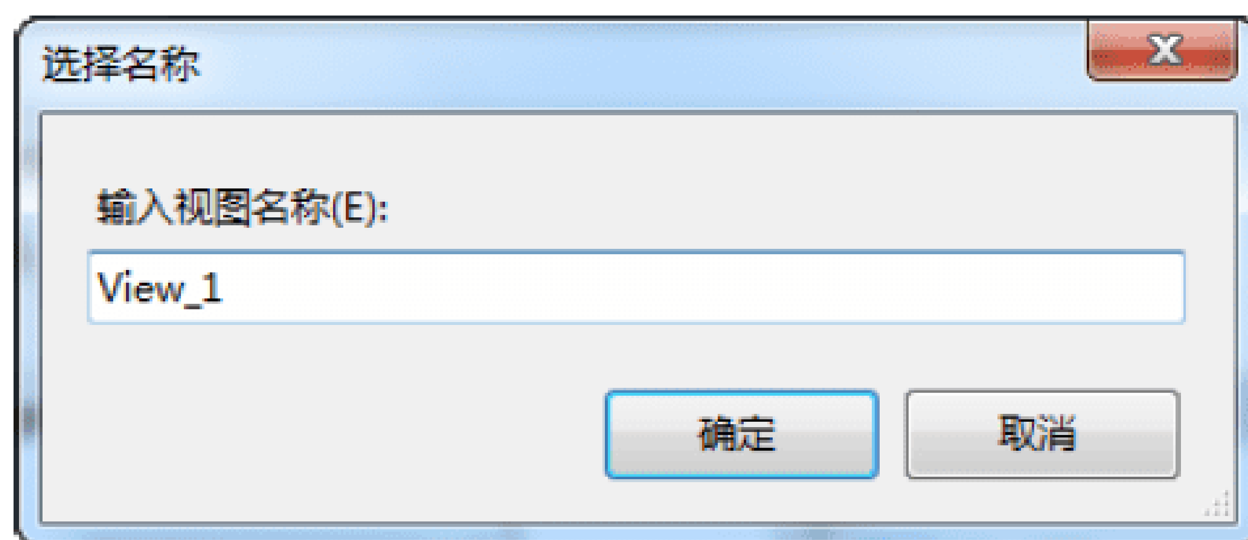


图 12-8　保存视图

（4）当需要重新修改视图设计时，右击视图名，在弹出的快捷菜单中，单击“修改”命令，按照新建视图的过程修改表设计后，单击“保存”按钮或“退出”按钮即可。

12.2.4　创建和管理存储过程

存储过程是执行一组数据库操作的代码集合，可以提高数据库的运行效率。在 SQL Server 2008 Management Studio 中创建存储过程的步骤如下。

（1）打开 SQL Server 2008 Management Studio，展开 SQL Server 组到要创建存储过程的数据库层，在“可编程性”|“存储过程”结点上右击，如图 12-9 所示。

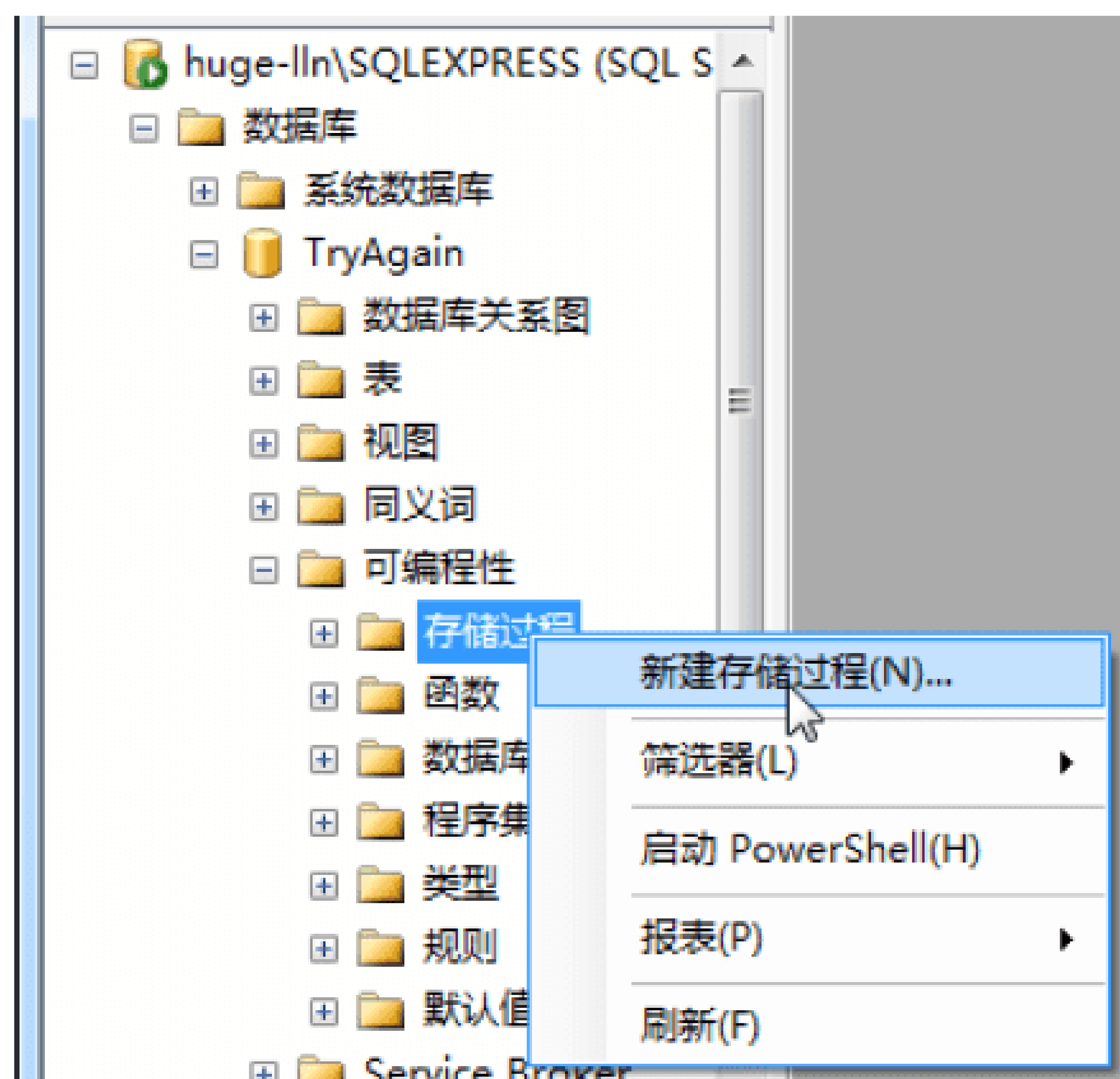


图 12-9　新建存储过程——第一步

（2）选择弹出菜单中的“新建存储过程”命令，打开“新建存储过程”对话框，如图 12-10 所示。

在文本框中输入创建存储过程的代码。单击工具栏上的分析按钮可以检查创建存储过程的代码是否有语法错误，单击另存为模板按钮可以将创建存储过程的 SQL 语句存储下来。

（3）输入完这些信息，单击工具栏中的执行按钮，存储过程就创建成功了。

（4）当需要重新修改存储过程时，右击存储过程名，在弹出的快捷菜单中选择“修改”

命令，按照新建存储过程的过程修改存储过程后，单击“保存”按钮或“退出”按钮即可。

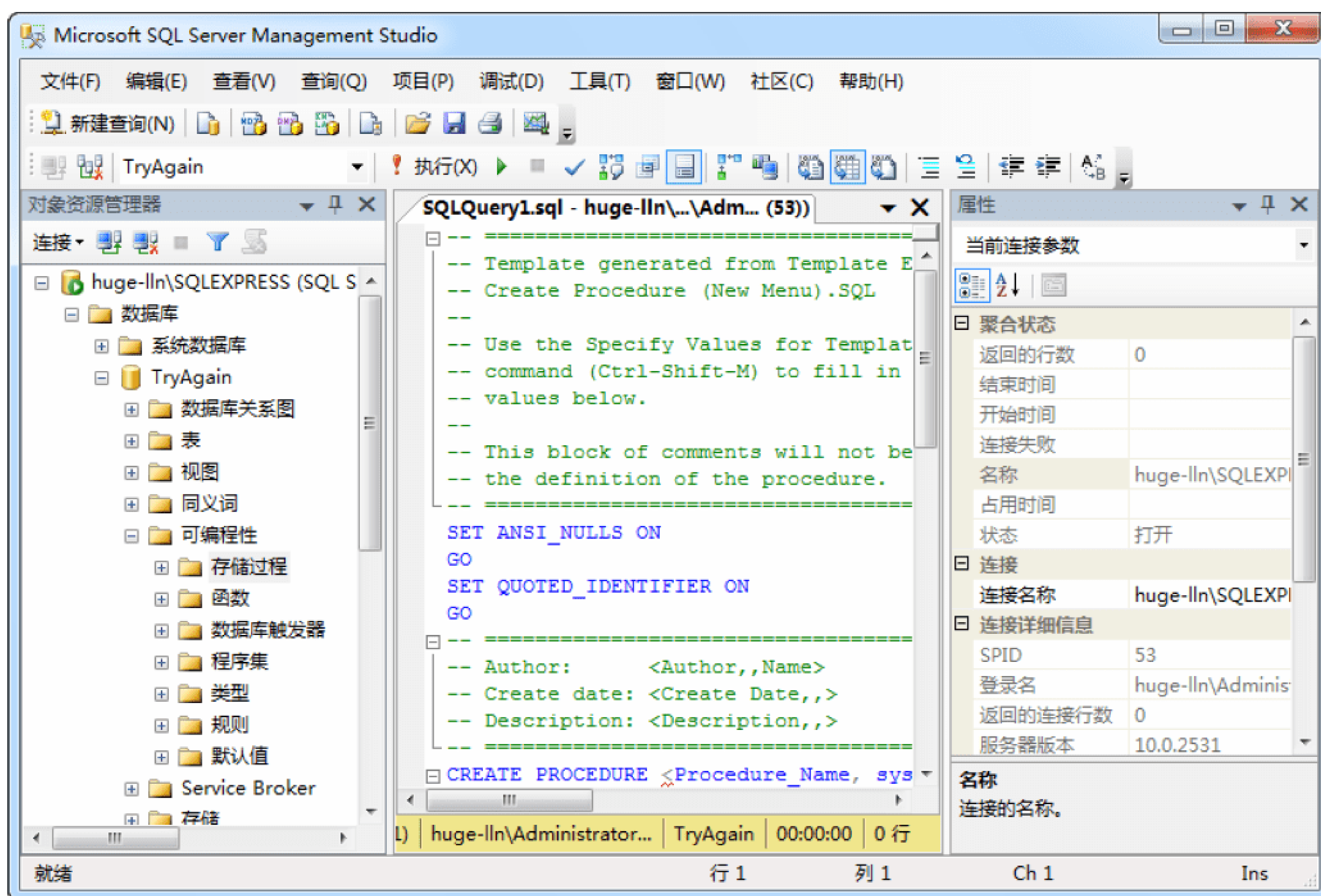


图 12-10　新建存储过程——第二步

12.3　ADO 访问技术

前面介绍了 SQL Server 2008 数据库及其使用，下面开始介绍在 VC 中如何使用 ADO 技术访问 SQL Server 数据库。本节首先介绍 ADO 模型和通过 ADO 技术访问数据库的具体步骤。

12.3.1　ADO 模型

ADO（ActiveX Data Objects），即 ActiveX 数据对象，是封装了 OLE DB 的 ActiveX 控件。OLE DB 是基于 COM 的通用数据访问技术，不仅支持索引顺序访问（ISAM）数据库和基于 SQL 的关系数据库，还支持其他数据源，可以从不同的数据源访问数据。不仅可以使用 SQL 查询获取数据，而且可以使用在提供程序中定义的查询。优点是使用方便、速度快、占用内存少。

ADO 的另一个功能是“远程数据访问”（RDS），能够通过传输将数据从服务器获取到客户端应用程序或 Web 页中，然后在客户端对数据进行操作，最后将数据更新到服务器。

ADO 模型主要由连接 Connection、命令 Command 和记录集 Recordset 组成。Connection 表示到数据源的一个连接，有关数据操作的对象都属于连接对象。除了具有描述自身属性的属性集合外，还有表示命令的 Command 对象，表示记录集的 Recordset 对象和存储错误

信息的 Errors 对象。Command 对象用于表示命令，除了具有描述自身属性的 Properties 集合外，还具有参数集合，用于存储命令所使用的参数的集合，其中包含多个参数对象。而记录集对象 Recordset 除了包含属性集合外，还具有描述返回的字段信息的字段集合 Fields，其中包含多个字段对象 Field。错误集合 Errors 中包含多个错误对象 Error，每个错误对象描述在操作过程中发生的错误。具体的模型图，如图 12-11 所示。

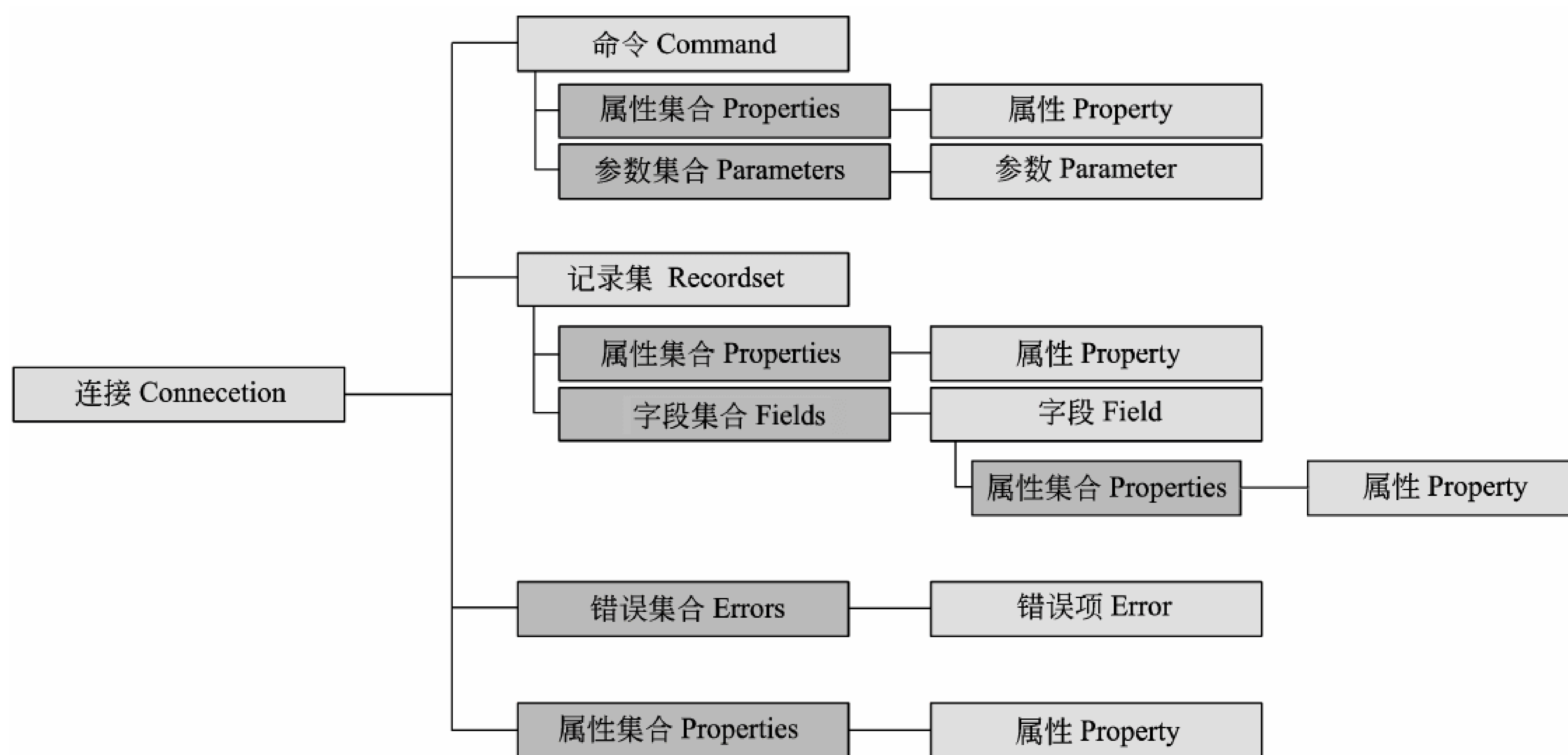


图 12-11　ADO 对象模型

在程序中使用 Connection 连接访问数据源，连接是访问数据必须使用的对象。ADO 模型，通过 Connection 对象实现事务的处理。所谓事务，是指将一系列数据访问操作看作一组操作，其中的任何一个操作失败，则整套操作失败，只有当全部操作步骤都成功，事务才算成功。

ADO 模型使用 Command 命令对象使命令对象化，提供命令优化，并用于具体操作数据源。命令可以在数据源中插入、删除或修改数据，也可以在表中检索数据。ADO 模型使用 Parameter 对象使参数对象化，在执行命令前，确定某些取值，之后，再使用这些取值执行命令。此处的参数与函数中的参数类似，如要在银行账务系统中进行转账，此时，可以将转入账户和转出账户作为参数，执行命令。

Recordset 记录集用于存储返回的数据查询集合。可查找、检索、插入、修改或删除记录，并通过其将对数据的更改回传到数据库中。ADO 模型使用 Field 对象使字段对象化，表示记录行的字段，每个字段具有名称、数据类型和值，其中值表示数据源中当前行的此字段的取值。

在执行数据库中的应用程序中随时可能发生错误，例如无法建立连接、执行命令失败等。ADO 模型使用 Error 对象表示错误。任意给定的错误都会产生一个或多个 Error 对象，随后产生的错误将会放弃先前的 Error 对象组。

上面所讲的每个 ADO 对象都有一组唯一的“属性”描述或控制对象的行为。属性有内置和动态两种类型。内置属性是 ADO 对象的一部分并且随时可用。动态属性则由特别的数据提供者添加到 ADO 对象的属性集合中，仅在提供者被调用时才存在。对象模型以

Property 对象体现属性。

ADO 提供“集合”，这是一种可方便地包含其他特殊类型对象的对象类型。使用集合方法可按名称（文本字符串）或序号（整型数）对集合中的对象进行检索。ADO 模型中提供了 4 种类型的集合：Connection 对象具有 Errors 集合，包含为响应与数据源有关的单一错误而创建的所有 Error 对象；Command 对象具有 Parameters 集合，包含应用于 Command 对象的所有 Parameter 对象；Recordset 对象具有 Fields 集合，包含所有定义 Recordset 对象列的 Field 对象；此外，Connection、Command、Recordset 和 Field 对象都具有 Properties 集合。它包含所有属于各个包含对象的 Property 对象。

除了上面介绍的对象，ADO 还引入了“事件”的概念。事件是对将要发生或已经发生的某些操作的通知。在程序中可以通过捕获事件，使用事件传入的参数进行处理。ADO 中主要有以下两种事件。

- Connection 事件：当连接中的事务开始、提交或回滚，Commands 执行时，Connections 开始或结束时产生的事件。在这些事件中，用户可以通过这些事件判断连接、命令或事务的执行结果。
- Recordset 事件：当在 Recordset 对象的记录行中进行定位，更改记录集行中的字段，更改记录集中的行，或在整个记录集中进行更改时，会触发此类事件。

12.3.2　ADO 数据库访问步骤分析

在 MFC 工程中，通过 ActiveX 控件可以使用可重用的数据绑定机制，快速地开发数据库应用程序。ADO 数据访问的步骤如下。

（1）使用 Connection 对象连接数据源，在此处可以选择事务支持。

（2）使用 Command 对象创建表示 SQL 命令的对象。

（3）使用 Parameter 对象设置指定列、表以及 SQL 命令中的值作为变量参数。

（4）使用 Command 对象、Connection 对象或 Recordset 对象执行命令。

（5）如果命令以行返回，将行存储在 Recordset 记录集对象中。

（6）使用 Recordset 对象创建存储对象的视图以便进行数据排序、筛选和定位。

（7）使用 Recordset 对象添加、编辑和删除数据。

（8）使用 Recordset 对象更新数据源。

（9）使用 Connection 对象可以完成事务的提交、事务取消和事务回滚。

上面这些步骤根据项目的实际需求，可能有的步骤不需要，有的步骤需要调整，读者可以根据自己的需求进行整合，编写出符合自己需求的数据库访问程序。

12.4　使用 ADO 访问数据库实例

本节介绍如何实现使用 ADO 访问数据库的常见功能：使用 ADO 连接 SQL Server 数据库、使用 ADO 读取数据库表记录以及使用 ADO 写入数据库表记录。本节以访问 SQL Server 数据为例，介绍使用 ADO 访问数据库的步骤。

12.4.1　ADO 连接 SQL Server 数据库

在使用 ADO 访问 SQL Server 数据库之前，首先需要在工程中增加对 ADO 访问技术的支持。步骤如下。

（1）按照前面介绍过的方法创建对话框应用程序 ADOSample。

（2）在 StdAfx.h 文件中增加如下代码，引入对 msado15.dll 的引用。

```
#import "c:\program files\common files\system\ado\msado15.dll"
no_namespace rename("EOF","adoEOF")     //引入 ADO 命名空间
```

（3）在应用程序类 CADOSampleApp 的 InitInstance()实例初始化函数中增加 COM 初始化语句，如下所示：

```
01  BOOL CADOSampleApp::InitInstance()          //初始化实例
02  {
03      AfxOleInit();                           //初始化 OLE
04  }
```

（4）在对话框类 CADOSampleDlg 中，增加表示数据库连接类和数据库记录集类的对象，代码如下：

```
 ConnectionPtr m pConnection;                //连接对象
_RecordsetPtr m_pRecordset;                  //记录对象
```

（5）做好如上的准备工作后，就可以使用 ADO 连接 SQL Server 数据库了，代码如下：

```
01  void CADOSampleDlg::OnButtonConnect()       //连接数据库
02  {
03      HRESULT hr;                             //定义操作结果句柄
04      try
05      {
06          hr = m_pConnection.CreateInstance("ADODB.Connection");
07          //创建 ADO 实例
08          if(SUCCEEDED(hr))               //如果创建实例成功，则打开数据库连接
09              hr = m pConnection->Open(
10              "provider=SQLOLEDB;Data Source=127.0.0.1;
11              Initial catalog=TryAgain;Integrated Security=SSPI;",
12              "sa","sa",adConnectUnspecified);
13          if (m pConnection->State)
14              WriteLog("数据库连接成功");
15          //如果创建成功，则提示成功
16          else
17              WriteLog("连接数据库失败");         //否则提示失败
18      }
19      catch(_com_error e)                     //连接数据库发生异常时的处理
20      {
21          CString log;                        //定义日志变量
22          log.Format("连接数据库失败!\r\n 原因:%s",e.ErrorMessage());
23          //显示日志信息
24          WriteLog(log);
25      }
26  }
```

上面代码首先调用 ADO 连接对象 m_pConnection 的 CreateInstance()函数，初始化连

接对象为 ADODB.Connection 类型。然后调用 ADO 连接对象 m_pConnection 的 Open()函数，打开到 SQL Server 数据库的连接，其中 Open()函数的参数依次指定连接字符串、连接用户名、连接密码和连接选项，此例中使用新创建的数据库 TryAgain。程序运行效果，如图 12-12 所示。当用户单击“连接数据库”按钮后，会在最下方的文本编辑框中显示操作结果。



图 12-12　连接数据库

12.4.2　ADO 读取数据库表记录

ADO 连接到数据库后，可以使用 ADO 模型中的记录集对象_RecordsetPtr 读取数据库表记录，读取数据后，还可以通过记录集对象在记录之间检索和处理数据，如定位记录、排序记录和过滤记录等操作。在读取记录时，可以设置字段的优化选项、记录查询的排序字段以及查询过滤器，可以使用记录集对象的 MoveFirst 成员函数移动到记录集第一条记录、使用 MoveNext()函数移动到记录集的下一条记录，并且可以处理当前记录的字段获取记录集中的数据。代码如下：

```
01  void CADOSampleDlg::OnButtonReadrecord()   //读取数据记录
02  {
03      if (m_pConnection == NULL)
04          return;       //如果数据库未连接成功，则返回
05      m_pConnection->CursorLocation =adUseServer;
06      //设置连接使用的光标类型
07      m_pRecordset.CreateInstance("ADODB.Recordset");
08      //创建记录集对象
09      m_pRecordset->Open("SELECT * FROM Employees",
10           variant_t((IDispatch*)m_pConnection,true),
11          //打开到数据库连接的记录
12          adOpenStatic,adLockOptimistic,adCmdText);
13      GetRecordContent();                          //获取记录集内容
14  }
```

上面代码用于读取 Employees 表中的所有数据，首先设置 ADO 连接对象的 CursorLocation 属性为 adUseServer，使得操作时使用服务器端的光标。然后调用记录集对象 m_pRecordset 的 CreateInstance()函数创建 ADODB.Recordset 类型的记录集，执行记录集

对象的 Open()函数，查询 Employees 表中的数据。从数据库服务器处查询到数据后，调用 GetRecordContent()自定义函数获取当前位置的记录的数据内容，并在对话框控件中显示出来。代码如下：

```
01  void CADOSampleDlg::GetRecordContent()//获取当前记录位置中的记录内容
02  {
03      //如果记录集对象为 NULL，则返回
04      if (m_pRecordset == NULL)
05          return;
06      //如果是记录集的头或尾，则返回
07      if ((m_pRecordset->BOF) || (m_pRecordset->adoEOF))
08          return;
09      _variant_t vID, vLastName, vFirstName;//记录字段变量
10      vID = m_pRecordset->GetCollect("EmployeeID");
11      //获取 EmployeeID 字段值
12      vLastName = m_pRecordset->GetCollect("LastName");
13      //获取 LastName 字段值
14      vFirstName = m_pRecordset->GetCollect("FirstName");
15      //获取 FirstName 字段值
16      m_ID = (LPCTSTR)(_bstr_t)(vID); //EmployeeID 字段值赋值为 ID 控件
17      m_LastName = (LPCTSTR)(_bstr_t)(vLastName);
18      //LastName 字段值赋值给控件
19      m_FirstName = (LPCTSTR)(_bstr_t)(vFirstName);
20      //FirstName 字段值赋值给控件
21      UpdateData(false);                //刷新控件显示的数据值
22  }
```

上面代码判断记录集对象是否为空，或是记录集当前位置是否到达记录集的开头和结尾，如果是，则无法获取记录数据并返回；否则，会调用记录集对象的 GetCollect()函数获取指定字段的值，并将其赋值给控件变量，最后调用 UpdateData()函数，将控件变量的值在控件中显示出来。此处只获取了员工编号和员工姓与员工名，要获取其他信息，使用同样的方法即可。下面的代码实现在记录中定位数据的功能。

```
01  void CADOSampleDlg::OnButtonFirst()     //移动到第一条
02  {
03      if (m_pRecordset == NULL)
04          return;                         //如果记录集对象为 NULL，则返回
05      m_pRecordset->MoveFirst();          //移动到记录集的第一条记录
06      GetRecordContent();                 //获取记录内容
07  }
08  void CADOSampleDlg::OnButtonLast()      //移动到最后一条
09  {
10      if (m_pRecordset == NULL)
11          return;                         //如果记录集对象为 NULL，则返回
12      m_pRecordset->MoveLast();           //移动到记录集的最后一条记录
13      GetRecordContent();                 //获取记录内容
14  }
15  void CADOSampleDlg::OnButtonPrev()      //移动到上一条
16  {
17      if (m_pRecordset == NULL)
18          return;                         //如果记录集对象为 NULL，则返回
19      if (m_pRecordset->BOF)
20          return;                         //如果是第一条记录，则返回
21      m_pRecordset->MovePrevious();       //移动到记录集的前一条记录
22      GetRecordContent();                 //获取记录内容
```

```
23  }
24  void CADOSampleDlg::OnButtonNext()       //移动到下一条
25  {
26      if (m_pRecordset == NULL)
27          return;                          //如果记录集对象为 NULL，则返回
28      if (m_pRecordset->EOF)
29          return;                          //如果是最后一条记录，则返回
30      m_pRecordset->MoveNext();            //移动到记录集的下一条记录
31      GetRecordContent();                  //获取记录内容
32  }
```

上面代码分别实现了移动记录到第一条、移动记录到最后一条、移动记录到上一条以及移动记录到下一条。在移动记录后，要记住获取当前记录中的数据，显示在界面上。当用户单击“读取数据记录”按钮时，会在下面的文本框中显示当前记录的数据，通过“第一条”按钮、“上一条”按钮、“下一条”按钮和“最后一条”按钮可以在记录集中导航数据。程序运行效果如图 12-13 所示。

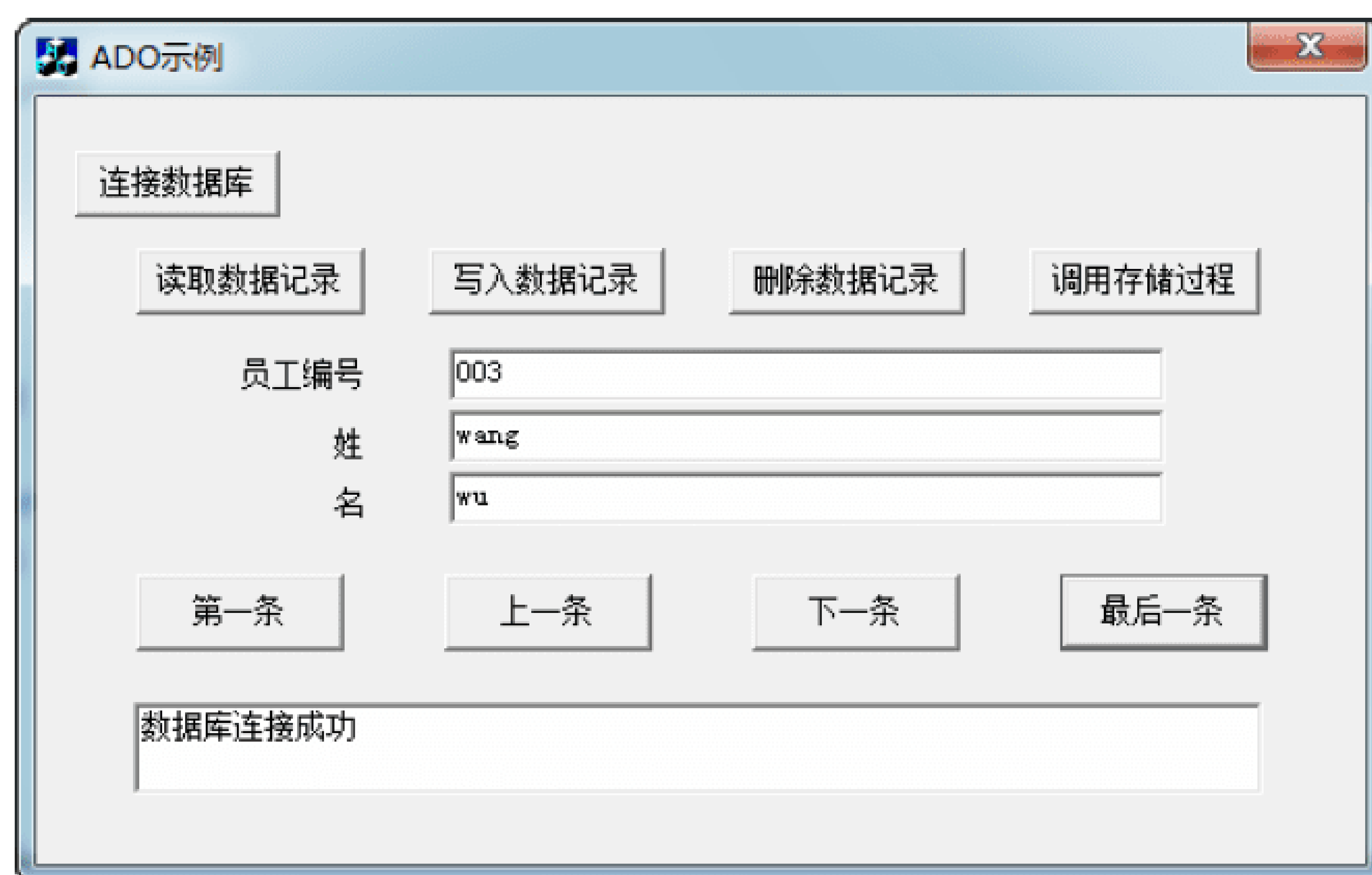


图 12-13　ADO 读取数据库记录运行效果

12.4.3　ADO 写入数据库表记录

要使用 ADO 写入数据库表记录，需要先调用连接对象的 BeginTrans()成员函数，开启事务，然后执行记录集的更新，最后调用连接对象的 CommitTrans()成员函数提交数据库更新。代码如下：

```
01  void CADOSampleDlg::OnButtonWriterecord()//写入数据记录
02  {
03      //如果链接对象或记录集对象为 NULL，则返回
04      if ((m_pConnection == NULL) || (m_pRecordset == NULL))
05          return;
06      try
07      {
08          UpdateData(true);                //获取数据控件中的数据
09          m_pRecordset->AddNew();          //调用 AddNew()函数增加记录
10          m_pRecordset->PutCollect("LastName", _variant_t(m_LastName));
11          //设置字段值
12          m_pRecordset->PutCollect("FirstName",_variant_tFirstName));
13          //设置字段值
```

```
14          m_pRecordset->Update();             //更新记录集
15          WriteLog("插入记录成功");            //显示操作日志
16      }
17      catch(_com_error e)                     //捕获错误异常
18      {
19          CString log;                        //日志信息变量
20          log.Format("插入记录失败!\r\n 原因:%s",e.ErrorMessage());
21          //格式化日志信息
22          WriteLog(log);                      //显示日志
23      }
24  }
```

上面代码首先调用 UpdateData()函数获取当前编辑框控件中的数据，然后调用记录集对象的 AddNew()方法，启动一次添加记录的工作，并调用 PutCollect()函数依次将要插入的字段的数据设置好，最后调用记录集对象的 Update()函数，即完成添加记录的功能。运行效果如图 12-14 所示。

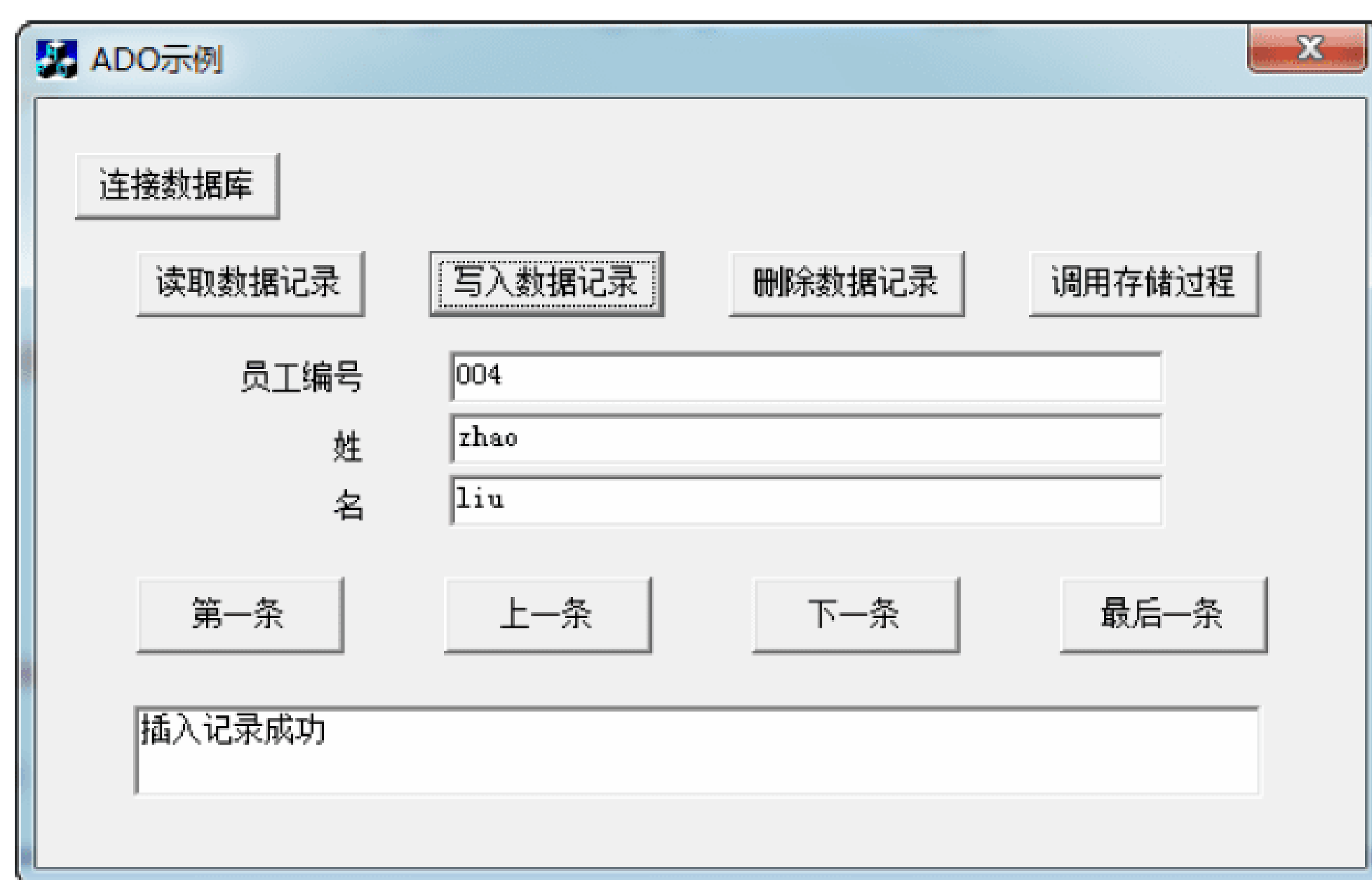


图 12-14　写入数据库记录的运行效果

12.4.4　ADO 删除数据库表记录

使用 ADO 可以删除数据库表中的记录，代码如下：

```
01  void CADOSampleDlg::OnButtonDelrecord2()    //删除记录
02  {
03      if (m_pRecordset == NULL)
04          return;  //如果记录集对象为 NULL，则返回
05      try
06      {
07          m_pRecordset->Delete(adAffectCurrent);
08          //调用 Delete()函数删除记录
09          m_pRecordset->Update();                 //更新记录集
10          m_pRecordset->Requery(NULL);            //重新查询记录集
11          GetRecordContent();                     //获取记录集内容
12          WriteLog("删除记录成功");                //显示操作日志
13      }
14      catch(_com_error e)                         //捕获错误异常
15      {
16          CString log;                            //日志信息变量
```

```
17          log.Format("删除记录失败!\r\n 原因:%s",e.ErrorMessage());
18          //格式化日志信息
19          WriteLog(log);                           //显示日志
20      }
21  }
```

上面代码调用记录集对象的 Delete()方法删除记录集当前位置的记录，并调用 Update()函数更新删除操作。删除完成后，重新调用查询函数刷新记录集对象中的数据，并调用 GetRecordContent()函数将记录集当前位置的数据显示在界面上。如图 12-15 所示，是删除数据库记录的运行效果。



图 12-15　删除数据库表记录的运行效果

12.5　本 章 小 结

本章主要介绍了 SQL Server 2008 的使用和 ADO 访问 SQL Server 数据库的方法。本章重点是理解 ADO 模型和使用 ADO 访问数据库的方法。第 13 章将介绍在 VC 中如何使用 ODBC 访问数据库。

12.6　习　　题

1．完成以下操作：

（1）创建名为 TASK 的数据库。

（2）在数据库 TASK 中创建一个表，命名为 January，字段包括 ID、name 和 time，并设置 ID 为主键。

【思路】按照 12.2.1 小节和 12.2.2 小节所介绍的步骤来完成即可。

2. 将 12.3.2 小节所讲的内容对应到 12.4 节实现的实例程序之中，体会使用 ADO 访问数据库的方法。

【思路】使用 Visual Studio 2010 打开 12.4 节的实例，然后找到各个步骤对应的程序代码，尝试修改来观察程序的运行效果。

第 13 章　Visual C++中 ODBC 访问技术

第 12 章讲述了 ADO 数据访问技术，虽然访问数据库的功能比较强大，而且开发也比较简单，但是对除 SQL Server 外的数据库的支持不完整并且也缺少性能优化。数据库的种类可谓是琳琅满目，各种数据库之间也是千差万别，但是其目的和功能都是一致的，就是管理数据。基于此，诞生了 ODBC（Open Database Connection）技术。本章将介绍在 VC 中如何使用 ODBC 技术。

13.1　ODBC API

在 VC 中可以调用 Windows 操作系统的底层 API，执行与系统有关的操作。同样地，在 VC 中可以直接调用 ODBC API 执行 ODBC 操作。ODBC API 是 ODBC 组件提供的数据库操作函数。本节将介绍有关 ODBC API 的基础知识。

13.1.1　ODBC 体系结构

ODBC 是一个调用级的接口，允许访问任何具有 ODBC 驱动的数据库。使用 ODBC 可以创建访问任何具有 ODBC 驱动的数据库的数据访问程序。ODBC 提供 API，允许应用程序独立于 DBMS。

ODBC 是 WOSA（Microsoft Windows Open Services Architecture，微软 Windows 开放服务系统）的标准数据库访问接口，允许基于 Windows 桌面应用程序，在每个平台上不用重写应用程序就可以连接到不同的数据库中。使用此接口可以不考虑各种 DBMS 的细节，只需要调用相应的 ODBC 驱动程序，即可完成对数据库的访问。ODBC 架构是基于客户端/服务器体系结构的，主要包含以下几部分。

- 数据源：是数据库和相关环境的结合，包括数据源运行的操作系统环境、数据库管理系统和网络结构等，使用数据源名称识别。数据源屏蔽掉了各种 DBMS 运行环境的差异，使用户以统一的角度处理对数据源的操作。要使用 ODBC API 类，数据源必须先通过 ODBC 管理器配置。对应用程序来说，可以访问任何具有 ODBC 驱动的数据源。
- ODBC API：是数据库访问应用编程接口，包含一组与数据库相关的函数和错误代码，并且 ODBC API 使用标准化查询语句 SQL 语句作为数据库查询语句。
- ODBC 驱动管理器：动态链接库 ODBC32.DLL，为应用程序装载和管理 ODBC 数据库驱动，并且支持同时加载和管理多个应用程序与多个驱动程序。此 DLL 对应用程序来说是透明的。

- ODBC 数据库驱动：处理指定 DBMS 的 ODBC 函数调用的一个或多个 DLL 集，负责执行 ODBC 函数调用，向数据源发送 SQL 请求，并将结果返回给应用程序。
- ODBC 光标库：动态链接库 ODBCCR32.DLL，在 ODBC 驱动中处理在数据中的导航。
- ODBC 管理器：用于配置 DBMS 的工具，使得数据源对应用程序可用。
- 应用程序：表示使用 ODBC 访问数据库的应用程序，SQL 命令由应用程序发送，并由应用程序处理返回的操作结果和错误。

比起直接访问 DBMS，使用 ODBC 访问 DBMS，程序可以独立于 DBMS。ODBC 驱动将调用转换成 DBMS 可以使用的命令，简化了开发人员的工作，使得数据源的可用范围变大。ODBC API 支持任何具有 ODBC 驱动的数据源，如关系数据库、ISAM 数据库、Microsoft Excel 电子表格或文本文件。ODBC 驱动管理从应用程序到数据源的连接，SQL 用于从数据库中选择记录。

13.1.2　ODBC 数据类型

ODBC 是数据库访问接口，因此提供对数据库中的数据类型的对应映射，在交换数据库中的数据和程序中的输入数据时，数据类型需要对应起来。表 13-1 列出了 ODBC 中的数据类型与 SQL 中的数据类型之间的对应关系。

表 13-1　ODBC数据类型与SQL数据类型之间的对应关系

ODBC数据类型	SQL数据类型	说　　明
CHAR	CString	SQL中的字符类型对应于CString类型
DECIMAL	CString	SQL中的小数类型对应于CString类型
SMALLINT	int	SQL中的整型对应于CString类型
REAL	float	SQL中的实数类型对应于float类型
INTEGER	long	SQL中的整型类型对应于long类型
FLOAT	double	SQL中的浮点类型对应于double类型
DOUBLE	double	SQL中的双精度类型对应于double类型
NUMERIC	CString	SQL中的数字类型对应于CString类型
VARCHAR	CString	SQL中的变字符类型对应于CString类型
LONGVARCHAR	CLongBinary、CString	SQL中的长变字符类型对应于CLongBinary或CString类型
BIT	BOOL	SQL中的位类型对应于BOOL类型
TINYINT	BYTE	SQL中的小整型类型对应于BYTE类型
BIGINT	CString	SQL中的长整型类型对应于CString类型
BINARY	CByteArray	SQL中的二进制类型对应于CByteArray类型
VARBINARY	CByteArray	SQL中的变长二进制类型对应于CByteArray类型
LONGVARBINARY	CLongBinary、CByteArray	SQL中的长二进制类型对应于CLongBinary或CbyteArray类型
DATE	CTime、CString	SQL中的日期类型对应于CTime或CString类型
TIME	CTime、CString	SQL中的时间类型对应于CTime或CString类型
TIMESTAMP	CTime、CString	SQL中的时间戳类型对应于CTime或CString类型

上面的数据类型在获取字段数据时，需要做好与程序中的数据类型的转换，也就是 C++数据类型与数据库数据类型，即 SQL 数据类型之间的对应。

13.1.3　ODBC 句柄与返回值

当使用 ODBC API 时，需要通过 ODBC 句柄存放 ODBC 对象，句柄是应用程序变量，使用句柄和 ODBC API 函数可以进行数据库操作。ODBC 主要分为以下 4 种句柄。

1．环境句柄SQLHENV

环境句柄 SQLHENV 用于表示 ODBC 中整个上下文，所有的 ODBC 应用程序在操作数据库前必须创建环境句柄，并在退出应用程序前释放 ODBC 环境句柄。可以用于管理其他类型的句柄，并且一个应用程序中，只能有一个环境句柄。以下是创建和释放环境句柄的代码。

```
01  SQLHENV sqlhenv1;                                //定义 SQL 环境句柄变量
02  //分配环境句柄
03  SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, & sqlhenv1);
04  SQLFreeHandle(SQL_HANDLE_STMT, * sqlhenv1);     //释放环境句柄
```

在上面代码中，第 1 行定义了类型为 SQLHENV 的环境句柄。第 3 行使用 SQLAllocHandle()函数初始化环境句柄。第 4 行调用 SQLFreeHandle()函数释放环境句柄。在释放环境句柄后，就不可以再调用 ODBC 函数了。

2．连接句柄SQLHDBC

连接句柄 SQLHDBC 用于管理有关数据库连接的信息。理论上，一个 ODBC 环境下，可以创建多个连接句柄，但是连接句柄是占用一定资源的，所以在分配连接句柄时要根据需要分配连接句柄。同时，ODBC 环境下支持的连接句柄的数目还与 ODBC 数据源中支持的连接句柄数目有关。如下是创建和释放 ODBC 连接句柄的代码。

```
01  //定义 SQL 连接句柄变量
02  SQLHDBC sqlhdbc1;
03  SQLAllocHandle(SQL_HANDLE_DBC, sqlhenv1, & sqlhdbc1); //分配连接句柄
04  SQLFreeHandle(SQL_HANDLE_DBC, * sqlhdbc1);            //释放连接句柄
```

上面代码首先声明 SQLHDBC 类型的连接句柄变量，然后调用 SQLAllocHandle()函数初始化连接句柄，传入的 3 个参数分别表示要初始化的句柄为连接句柄、与连接句柄关联的环境句柄和存储连接句柄的变量。在初始化连接句柄时，驱动管理器会分配一个存储有关语句信息的结构，并在变量中返回连接句柄。为了节约资源，在使用完连接句柄后，需要调用 SQLFreeHandle()函数释放连接句柄。

3．语句句柄SQLHSTMT

语句句柄 SQLHSTMT 用于处理 SQL 语句和结构函数，语句句柄是与连接句柄相关联的，当应用程序发送调用指令到驱动程序时，ODBC 驱动管理器会将语句句柄中的指令发送给与语句句柄关联的连接句柄，由其发送给相应的驱动程序。以下是创建和释放 ODBC

语句句柄的代码。

```
01  SQLHSTMT  hstmt;                                    //定义 SQL 语句句柄变量
02  SQLAllocHandle(SQL_HANDLE_STMT,hdbc, &hstmt)       //分配语句句柄
03  SQLFreeHandle(SQL_HANDLE_STMT, * hstmt);           //释放语句句柄
```

上面代码首先声明 SQLHSTMT 类型的表示 ODBC 语句句柄的变量，然后调用 SQLAllocHandle()函数初始化语句句柄，传入的 3 个参数分别表示要初始化的句柄为语句句柄、与语句句柄关联的连接句柄和存储语句句柄的变量。为了节约资源，在使用完语句句柄后，需要调用 SQLFreeHandle()函数释放语句句柄。

4. 描述器句柄DESCRIPTOR

描述器句柄 DESCRIPTOR 用于描述元数据的信息，主要包括描述 SQL 语句的参数和数据库表的列信息等。默认情况下，初始化语句句柄后，会自动生成描述器。也可以在应用程序中手动分配描述器句柄。

在调用 ODBC API 函数时，统一使用 SQLRETURN 类型表示函数的执行结果。其中，SQL_SUCCESS 表示函数操作成功；SQL_SUCCESS_WITH_INFO 表示函数操作结果中带有警告信息；SQL_ERROR 表示函数操作失败。以下为调用 ODBC API 函数的错误处理的推荐方式。

```
01  SQLRETURN rtCode;   //定义返回值
02  rtcode=XXXXXX       //执行操作记录返回值
03  //如果返回值为带警告的信息，但是成功，则处理
04  if(rtCode ==SQL_SUCCESS_WITH_INFO)
05  {//处理警告信息}
06  else
07  {//处理出错信息}        //如果函数失败，则处理错误信息
08  //函数调用成功，程序继续进行
09  ...
```

13.1.4　ODBC 驱动和管理器

ODBC 驱动器用于注册和配置可用的本地或网络数据源。要使用 ODBC 数据源，首先需要注册和配置 ODBC 数据源，使用 ODBC 管理器可以增加和删除数据源，并且可以根据 ODBC 驱动创建新数据源。ODBC API 使用 ODBC 管理器提供的信息创建程序中连接用户数据源的代码。

要使用 ODBC 编写数据库应用程序，必须安装 ODBC 管理器和相应数据源的驱动。默认情况下，在安装操作系统时会自动安装 ODBC 管理器和部分 ODBC 数据源驱动。如果要安装新 ODBC 数据源驱动，则需要运行与驱动相连的 setup 程序。

驱动安装好后，就可以使用 ODBC 管理器配置数据源。此时在“控制面板”中，就出现了 ODBC 图标，读者就可以使用 ODBC 管理器管理 ODBC 数据源。VC 可以提供 ODBC 驱动的数据库包括 SQL Server、Microsoft Access、Microsoft FoxPro、Microsoft Excel、dBASE、Paradox、Microsoft Oracle ODBC 和文本文件。

13.1.5　配置 ODBC 数据源

应用程序要使用数据源，必须使用 ODBC 管理器配置 ODBC 数据源。ODBC 管理器在 Windows 注册表中记录数据源和其连接信息。使用 ODBC 管理器可以在数据源对话框中增加、修改和删除数据源，并且可以增加和删除 ODBC 驱动。配置 ODBC 数据源的步骤如下。

（1）选择“开始”|“所有程序”|“控制面板”|“管理工具”|“数据源（ODBC）”命令，打开“ODBC 数据源管理器”对话框，如图 13-1 所示。

（2）在图 13-1 中，选择“系统 DSN”选项卡，并单击“添加”按钮，打开“创建新数据源”对话框，如图 13-2 所示。

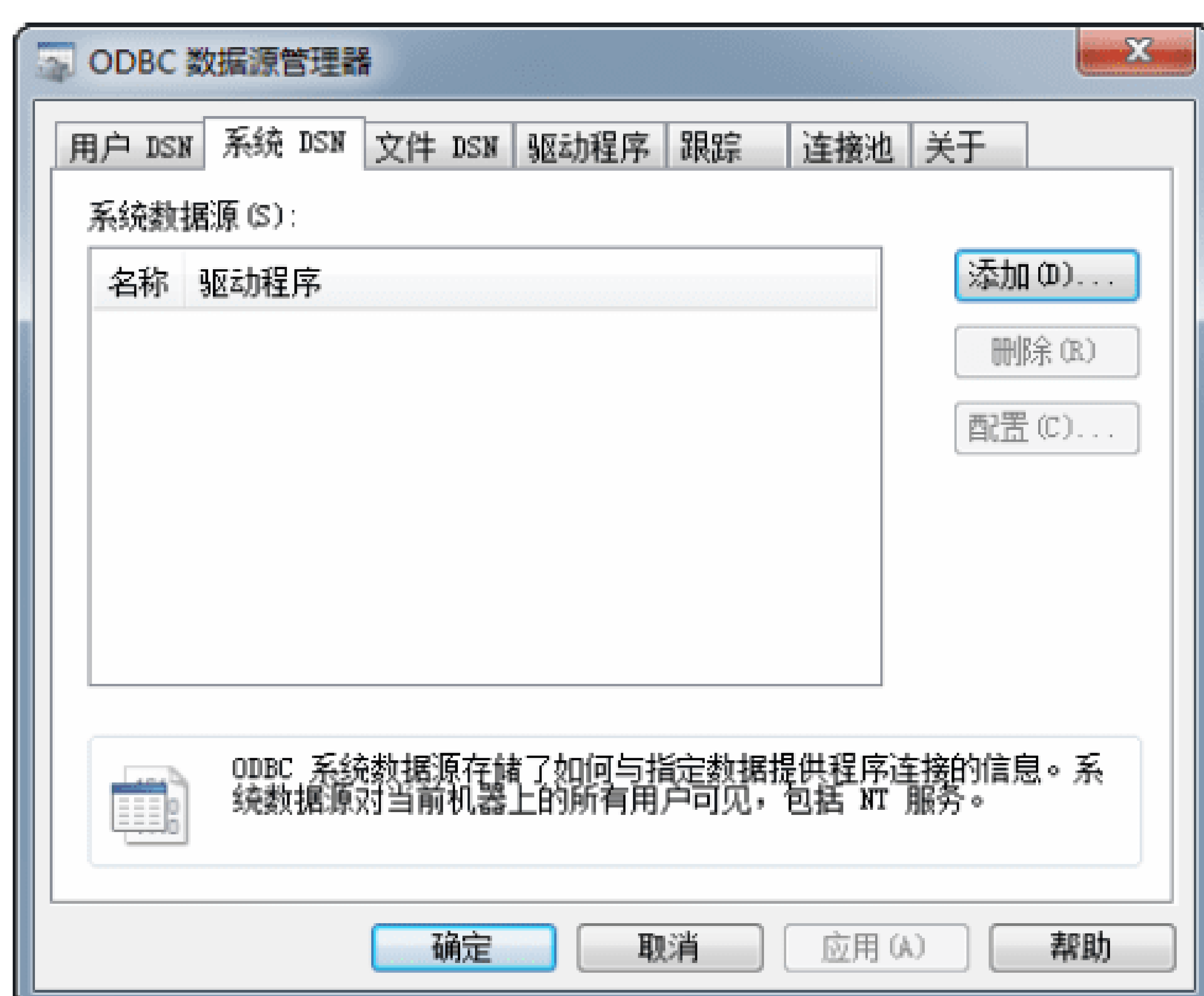


图 13-1　“ODBC 数据源管理器”对话框

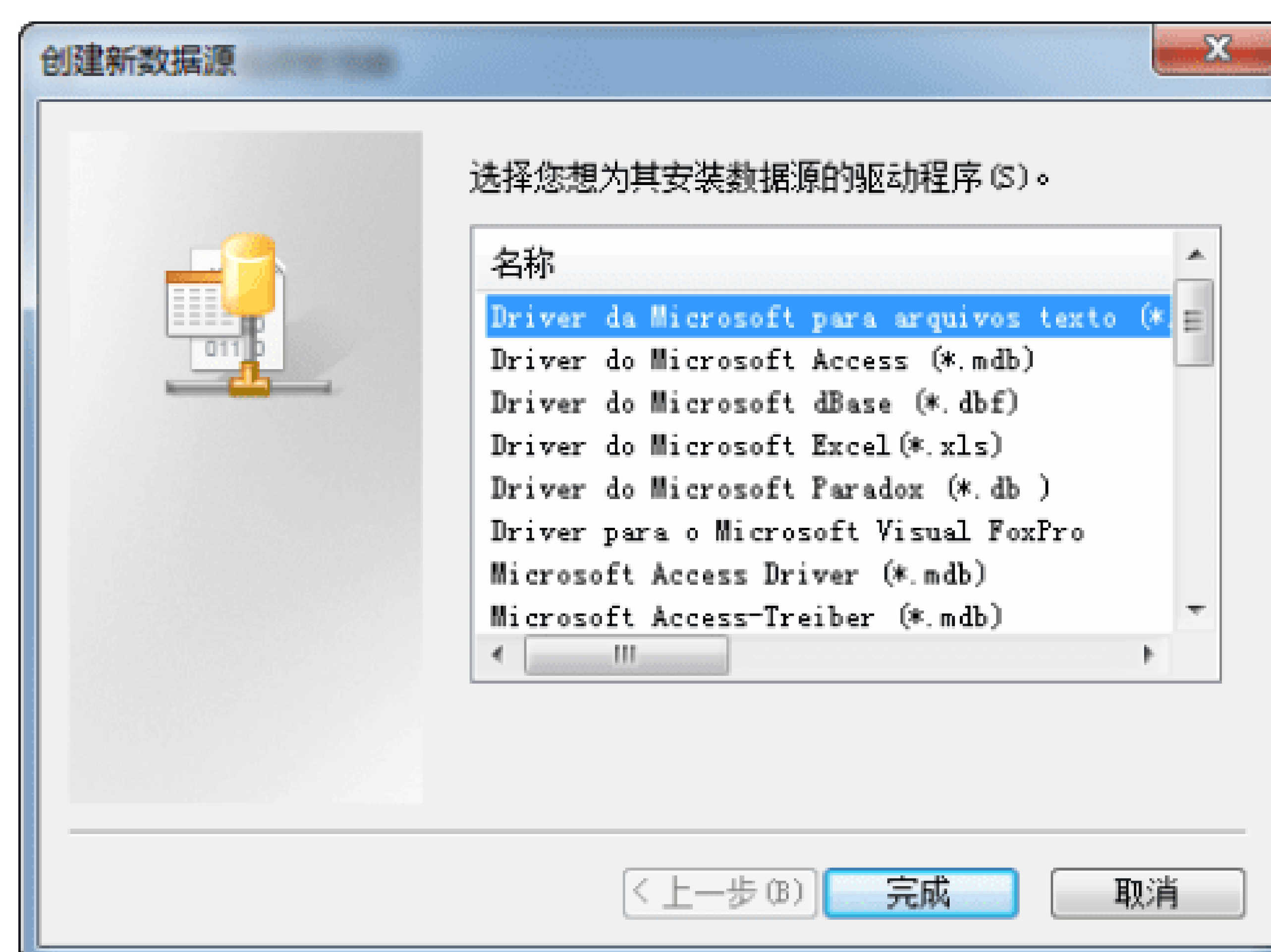


图 13-2　“创建新数据源”对话框

（3）在图 13-2 所示的列表框中，选择要创建的 ODBC 数据源使用的数据库引擎，本例中选择 SQL Server 表示要创建的 ODBC 数据源要连接的 DBMS 是 SQL Server 数据库，单击“完成”按钮，打开“创建到 SQL Server 的新数据源”对话框，如图 13-3 所示。

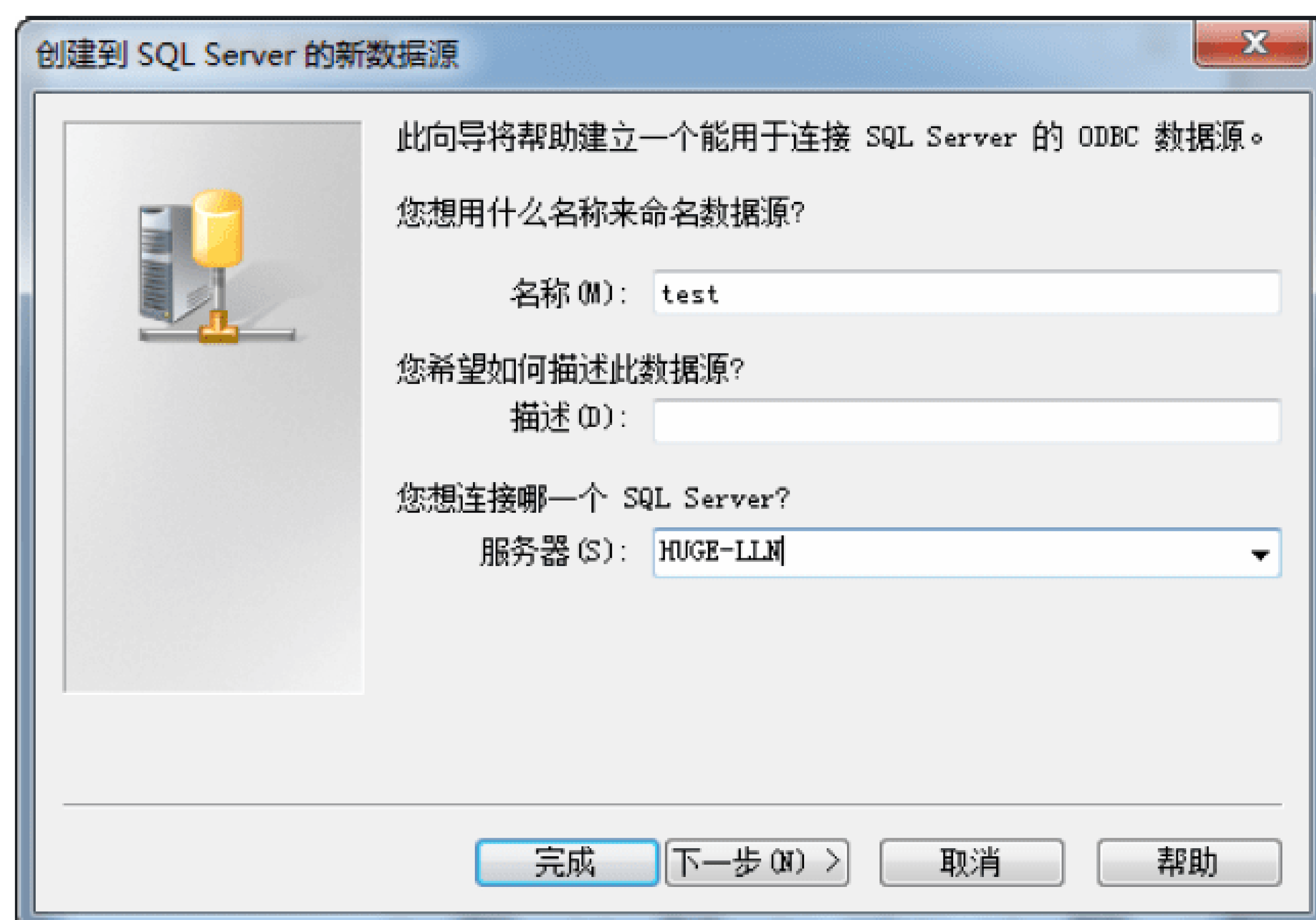


图 13-3　“创建到 SQL Server 的新数据源”对话框

（4）在图 13-3 所示的“名称”文本框中输入要创建的数据源的名称，在“描述”文本

框中输入数据源的描述信息，在“服务器”文本框中选择输入要连接的数据库服务器，单击“下一步”按钮，打开登录信息输入对话框，如图 13-4 所示。

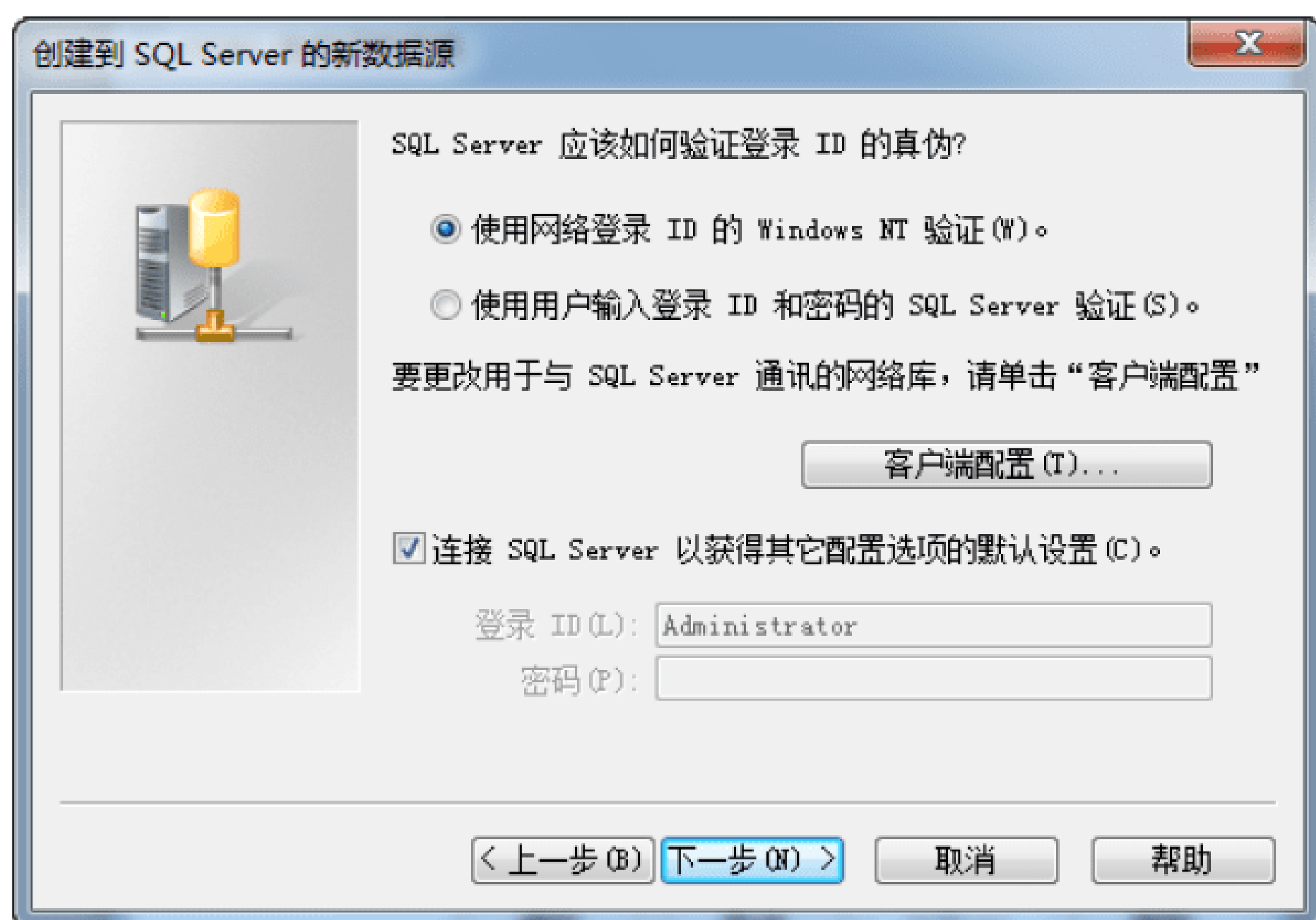
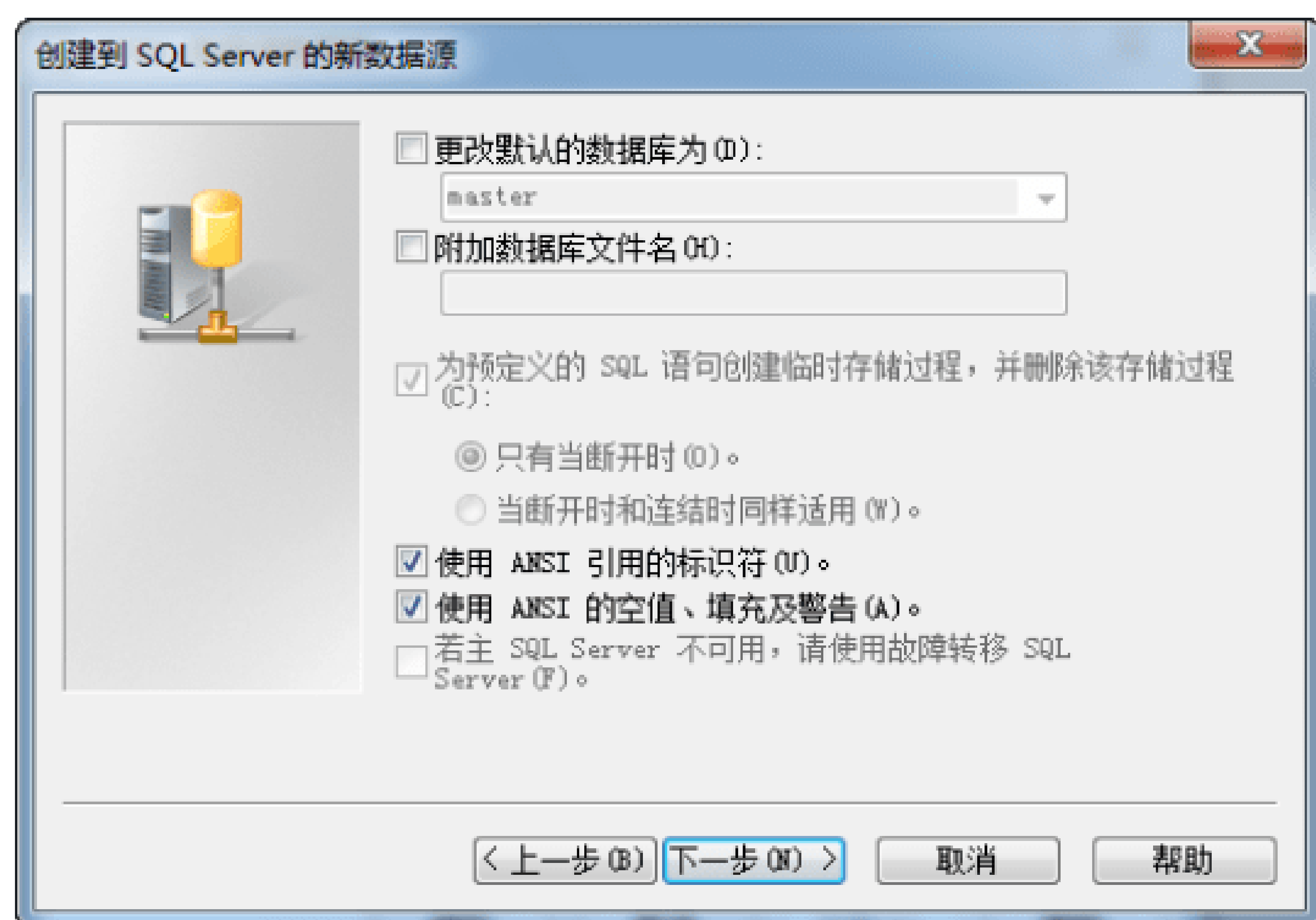


图 13-4　登录信息输入对话框

（5）在图 13-4 中，选择“使用网络登录 ID 的 Windows NT 验证”单选按钮，单击“下一步”按钮，打开数据库设置对话框，如图 13-5 所示。

（6）在图 13-5 中选择“更改默认的数据库为”复选框，并在下拉列框中选择要连接的数据库名称，单击“下一步”命令，打开其他库设置对话框，如图 13-6 所示。



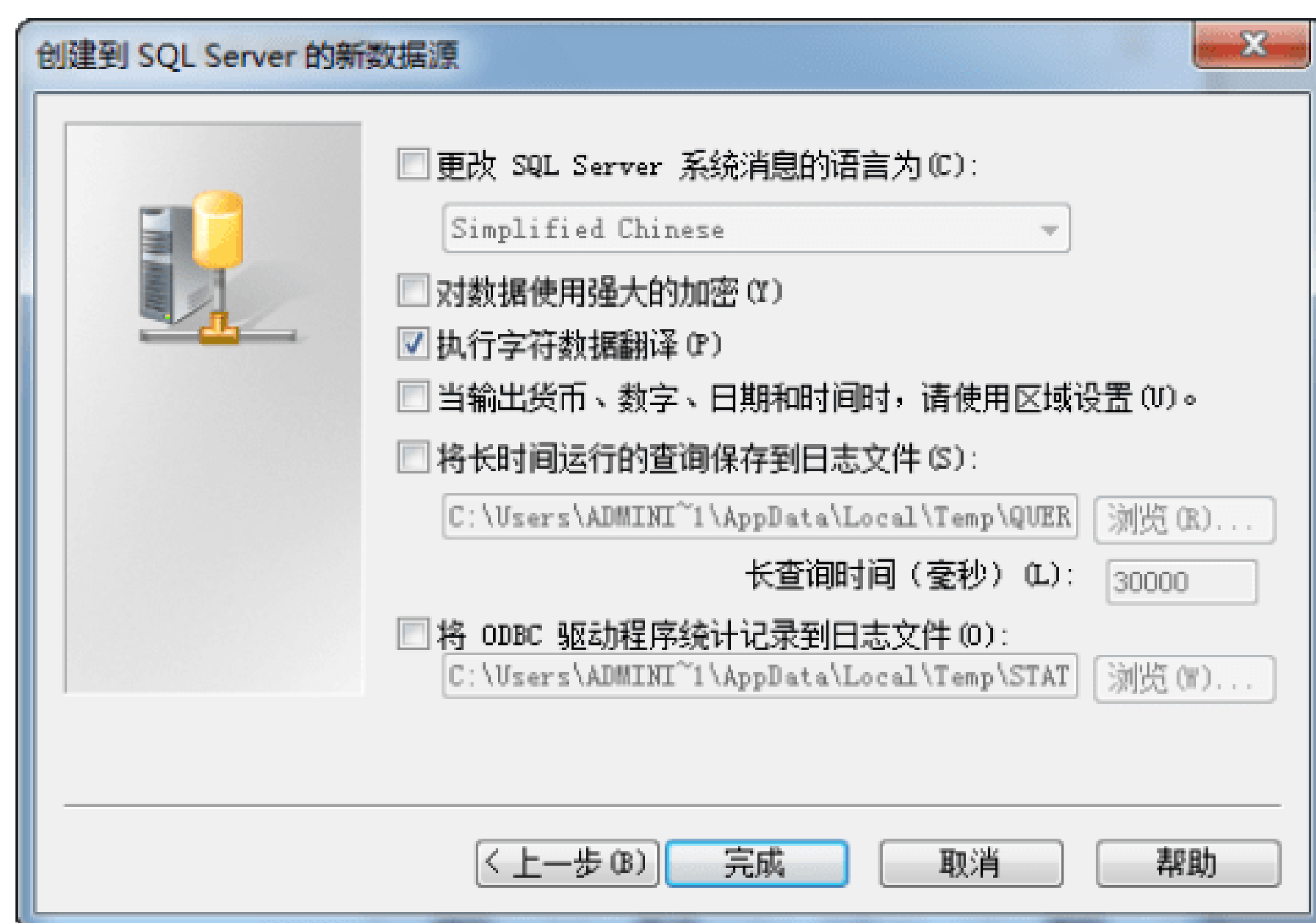


图 13-5　数据库设置对话框　　　图 13-6　其他库设置对话框

（7）在图 13-6 中，选择“更改 SQL Server 系统消息的语言为”复选框，选择使用的语言，单击“完成”按钮，打开“ODBC Microsoft SQL Server 安装”对话框，如图 13-7 所示。

（8）图 13-7 显示了设置的参数，单击“测试数据源...”按钮，测试配置的 ODBC 数据源是否连接成功，打开“SQL Server ODBC 数据源测试”对话框，如图 13-8 所示。

（9）当在图 13-8 所示的信息框中出现“测试成功！”提示时，表示配置的 ODBC 数据源成功。单击“确定”按钮后，返回配置界面，单击“确定”按钮，这样会返回到“ODBC 数据源管理”对话框中，并在列表框中增加了一条刚刚配置成功的 ODBC 数据源信息。

使用上面的步骤就完成了 ODBC 数据源的配置。需要注意的是，上面是以连接 SQL Server 数据库为例配置的 ODBC，当要连接其他数据源时，配置过程会有不同，具体情况

需要根据使用的 DBMS 数据源类型而定。



图 13-7　ODBC 安装对话框



图 13-8　ODBC 数据源测试对话框

13.2　用 ODBC API 操作数据库实例

使用 ODBC API 函数可以完成所有 ODBC 支持的有关数据库的操作，包括数据检索功能、数据库目录函数等，并且使用 ODBC API 编写出来的程序简洁高效。本节将介绍使用 ODBC API 操作数据库的方法。

13.2.1　操作数据库的一般步骤

使用 ODBC API 操作数据库有一系列的步骤，按照此步骤可以完成对数据库的操作。在执行每步操作时，需要判断前一步的返回结果。如果前一步操作失败，应该中断下面的过程，否则，可能会在后面的操作使用无效句柄，造成程序发生异常。步骤如下。

（1）分配 ODBC 环境句柄：任何 ODBC 应用程序，必须首先装载 ODBC 驱动管理器，并使用 ODBC 环境句柄中介绍的方法初始化 ODBC 环境句柄。

（2）分配连接句柄：要操作数据库，必须创建与数据库相连的连接句柄，方法在介绍连接句柄时介绍过。一个连接句柄对应于一个数据源，并与环境句柄相连。

（3）连接数据源：分配完连接句柄后，就可以设置连接属性，调用连接函数建立与 ODBC 数据源的连接。

（4）构造和执行 SQL 语句：在此处可以根据应用需求，构造和执行相应的 SQL 语句，这部分是操作数据库的核心部分，使用 ODBC API 既可以执行查询操作，也可以执行增加、修改和删除操作，具体方法要根据实际需求而定。

（5）处理操作结果：执行完 SQL 语句后，应用程序根据需要处理操作结果。

（6）断开数据源连接：执行完数据库操作后，为了节省系统资源，需要断开与数据源的连接。

（7）释放 ODBC 环境句柄：对所有的数据库的访问完成后，为了节省系统资源，需要

释放 ODBC 环境句柄，并释放程序中使用的存储空间。

13.2.2　连接数据库

前面介绍过，连接数据库前需要根据需求设置连接属性，一般在设置连接属性前，会首先使用 SQLGetConnectAttr()函数获取当前的属性设置，然后根据需求调用 SQLSetConnectAttr()函数设置属性，最后调用连接函数连接数据库。以下代码是 SQLGetConnectAttr()函数和 SQLSetConnectAttr()函数的原型。

```
SQLRETURN SQLGetConnectAttr(
    SQLHDBC ConnectionHandle,   //ODBC 数据库连接句柄
    SQLINTEGER Attribute,       //指定要获取的属性
    SQLPOINTER ValuePtr,        //指定要获取的属性值
    SQLINTEGER StringLength);   //获取属性值的长度
SQLRETURN SQLSetConnectAttr(
    SQLHDBC ConnectionHandle,   //ODBC 数据库连接句柄
    SQLINTEGER Attribute,       //指定要设置的属性
    SQLPOINTER ValuePtr,        //指定要设置的属性值
    SQLINTEGER StringLength);   //设置属性值的长度
```

完成数据库属性设置后，就可以连接数据库了。ODBC API 中提供了 3 个用于连接数据库的函数，分别为 SQLConnect()函数、SQLDriverConnect()函数和 SQLBrowseConnect()函数。其中，SQLConnect()函数是最简单的连接函数，使用此函数，只需要提供 ODBC 数据源名称、用户名和密码就可以连接数据库。其函数原型为：

```
SQLRETURN SQLConnect(
    SQLHDBC ConnectionHandle,       //ODBC 数据库连接句柄
    SQLCHAR *ServerName,            //指定要连接的 ODBC 数据源的名称
    SQLSMALLINT NameLength1,        //指定要连接的 ODBC 数据源的名称的长度
    SQLCHAR *UserName,              //指定要连接 ODBC 数据源的用户名
    SQLSMALLINT NameLength2,        //指定要连接 ODBC 数据源的用户名的长度
    SQLCHAR *Authentication,        //指定要连接 ODBC 数据源的密码
    SQLSMALLINT NameLength3);       //指定要连接 ODBC 数据源的密码的长度
```

SQLDriverConnect()函数是使用连接字符串提供更多连接参数的连接函数。其函数原型为：

```
SQLRETURN SQLDriverConnect(
    SQLHDBC  ConnectionHandle,      //ODBC 数据库连接句柄
    SQLHWND  WindowHandle,          //指定对话框句柄，通常是应用程序的对话框句柄
    SQLCHAR *InConnectionString,    //指定要连接的 ODBC 的连接字符串
    SQLSMALLINT StringLength1,      //指定要连接的 ODBC 的连接字符串的长度
    SQLCHAR *OutConnectionString,   //存放连接成功后完整的连接字符串
    SQLSMALLINT BufferLength,       //指定 OutConnectionString 参数的大小
    SQLSMALLINT  *StringLength2Ptr, //存放连接成功后完整的连接字符串的长度
    SQLUSMALLINT DriverCompletion);//指定是否需要驱动管理器提供更多的连接信息
```

SQLBrowseConnect()函数提供迭代方式到数据源的连接，是基于客户端/服务器架构的，因此，本地数据库不支持此种方式。具体使用哪个连接数据源函数，可以根据需求自

已选择。

13.2.3　读取数据库表记录

应用程序对数据库的操作主要通过 SQL 语句实现，而 SQL 语句在 ODBC API 中通过语句句柄表示，因此首先需要初始化语句句柄，然后调用 SQLExecute()函数执行语句句柄代表的 SQL 语句。SQLExecute()函数用于执行准备好的 SQL 语句。其函数原型为：

```
SQLRETURN SQLExecute(
    SQLHSTMT StatementHandle);  //表示要执行的 SQL 语句的句柄
```

SQLExecDiret()函数用于直接执行 SQL 语句，一般用于只需要执行一次的操作，此函数执行 SQL 语句的效率是最高的。其函数原型为：

```
SQLRETURN SQLExecDirect(
    SQLHSTMT StatementHandle,               //语句句柄
    SQLCHAR *StatementText,                 //要执行的 SQL 语句
    SQLINTEGER  TextLength);                //要执行的 SQL 语句的长度
```

SQLExecDiret()函数执行只需要执行一次的函数效率较高，对于需要执行多次的 SQL 语句，则执行前使用 SQLPripare()函数先预处理 SQL 语句，可以提高程序的运行速度。其函数原型为：

```
SQLRETURN SQLPrepare(
    SQLHSTMT StatementHandle,       //语句句柄
    SQLCHAR* StatementText,         //要执行的 SQL 语句
    SQLINTEGER TextLength);         //要执行的 SQL 语句的长度
```

在执行完 SQL 语句后，可以通过 ODBC API 中提供的函数检索和处理返回的结果数据。首先需要使用 SQLBindCol()函数绑定结果集中的列到应用程序变量上，这样才可以通过程序变量获取返回的结果。其函数原型为：

```
SQLRETURN SQLBindCol(
    SQLHSTMT  StatementHandle,      //语句句柄
    SQLUSMALLINT  ColumnNumber,     //指定要绑定的列的序号
    SQLSMALLINT  TargetType,        //指定绑定的列的数据类型
    SQLPOINTER TargetValuePtr,      //指定绑定到列的数据缓冲区的指针
    SQLINTEGER  BufferLength,       //指定缓冲区的长度
    SQLINTEGER  *StrLen_or_IndPtr); //指定缓冲区使用的长度的指针
```

使用 SQLBindCol()函数绑定到返回的结果记录集列后，就可以使用 SQLFetch()函数检索记录集数据。其会将光标移动到记录集中的下一条记录，并将所有绑定的列的数据复制到绑定时指定的程序变量中。其函数原型为：

```
SQLRETURN SQLFetch(SQLHSTMT StatementHandle)   //语句句柄
```

将上面介绍的这几个函数组合起来就可以实现从数据库表中读取记录的功能。由于数据库应用需求的千差万别，所以需要用户根据实际需求，使用适当的语句编写符合需求的应用程序。

13.2.4　添加、删除记录

有 3 种方式可以实现添加和删除记录。一种是在连接句柄上直接执行相应的 SQL 语句；另一种是调用 SQLSetPos()函数实现更新记录集定义；还有一种是调用 SQLBulkOperations()函数实现。后两种方式根据数据源的不同，有的是不支持的。

SQLBulkOperations()函数用于在当前行集上执行更新操作，在调用其进行更新操作前，必须首先调用 SQLFetch()函数获取行集。其函数原型为：

```
SQLRETURN SQLBulkOperations(
    SQLHSTMT StatementHandle,   //语句句柄
    SQLUSMALLINT Operation);    //表示要执行的更新操作
```

其中，有效更新操作类型有 SQL_ADD、SQL_UPDATE_BY_BOOKMARK、SQL_DELETE_BY_BOOKMARK 和 SQL_FETCH_BY_BOOKMARK，分别表示增加记录、修改记录、删除记录和定位记录。

要添加或删除记录，需要传入不同的参数执行 SQL 语句，如增加一条新的员工记录使用 SQLBindParameter()函数可以实现为 SQL 语句传入参数的功能。其函数原型为：

```
SQLRETURN  SQLBindParameter(
    SQLHSTMT  StatementHandle,          //语句句柄
    //指定要绑定的参数在 SQL 中的序号，从 1 开始编号
    SQLUSMALLINT  ParameterNumber,
    SQLSMALLINT  InputOutputType,       //指定要绑定的参数类型
    SQLSMALLINT  ValueType,             //指定参数值的类型
    SQLSMALLINT  ParameterType,         //指定参数数据类型
    SQLUINTEGER  ColumnSize,            //指定参数值的大小
    SQLSMALLINT  DecimalDigits,         //指定参数精度
    SQLPOINTER  ParameterValuePtr,      //指向存放参数值的缓冲区的指针
    SQLINTEGER  BufferLength,           //指定存放参数值的缓冲区的大小
    SQLINTEGER  *StrLen_or_IndPtr); //指向 ParameterValutePtr 参数的指针
```

其中，InputOutputType 参数用于指定要绑定的参数类型，有效取值有 SQL_PARA_INPUT、SQL_PARAM_INPUT_OUTPUT 和 SQL_PARAM_OUTPUT，分别表示输入参数、输入输出参数和输出参数。ParameterType 参数指定参数数据类型。

如果要操作数据量较大的文本文档或位图文件，则需要分开传递参数值。下面两个函数可以设置指定语句句柄对应的参数值，函数原型为：

```
SQLRETURN SQLPutData(
    SQLHSTMT StatementHandle,           //语句句柄
    SQLPOINTER DataPtr,                 //指向存放参数值的缓冲区的指针
    SQLINTEGER StrLen or Ind);          //指定参数值缓冲区的长度
    //语句句柄
    SQLRETURN SQLParamData(SQLHSTMT StatementHandle,
    SQLPOINTER *ValuePtrPtr);           //指向缓冲区地址的指针
```

13.2.5　断开数据库连接

当完成数据库操作后，不要忘记断开与数据库的连接。如果使用完后不做清理工作，

则 ODBC 数据库连接池中的连接个数会不断增长，从而影响程序效率，同时，当数据库连接池中的连接数目达到一定数目后，会导致其他程序无法连接到数据库。ODBC API 中使用 SQLDisconnect()函数断开与数据库的连接。其函数原型为：

```
SQLRETURN SQLDisconnect(SQLHDBC ConnectionHandle);//要断开的连接句柄
```

上面的函数会断开到 ODBC 数据源的连接。如果要断开的数据库连接中有事务未完成，则函数返回的 SQLSTATE 的值为 25000。要获取 SQLSTATE 的值，可以通过 SQLGetDiagRec()函数获取，其用于获取具体的错误信息。最后不要忘记释放环境句柄。

13.2.6　ODBC API 封装类实例

前面几小节介绍了如何使用 ODBC API 操作数据库，本小节以一个实例，讲解如何使用 ODBC API 来操作数据库。在实际使用 ODBC API 操作数据库时，通常将其封装在类中，本小节封装一个客户类，可以实现查询客户名称和联系人名称、添加客户信息和删除客户信息的功能。代码如下：

```
01  class MyODBCAPI                    //ODBC API 类封装
02  {
03  public:
04      MyODBCAPI() {InitODBC();}      //构造函数
05      ~MyODBCAPI();                  //析构函数
06      SQLHENV henv;                  //环境句柄
07      SQLHDBC hdbc;                  //连接句柄
08      SQLHSTMT hstmt;                //语句句柄
09      SQLRETURN retcode;             //错误代码
10      CString msg;                   //当前错误信息
11      BOOL bInit;                    //是否初始化成功
12      BOOL InitODBC();               //初始化 ODBC 数据源
13      //连接 ODBC 数据源
14      BOOL Connect(CString odbcName, CString userID, CString pass);
15      BOOL ExecSQL(CString sql); //执行 SQL 语句
16      void QueryCustomer();          //查询客户数据
17      //插入客户信息
18      void InsertCustomer(CString CompanyName, CString ContactName);
19      //删除客户信息
20      void DeleteCustomer(CString CustomerID);
21  };
```

上面代码是 ODBC API 封装类的定义，其中定义了环境句柄 henv、连接句柄 hdbc、语句句柄 hstmt 和错误代码 retcode 以及错误信息 msg。其中 bInit 变量用于记录当前是否初始化 ODBC 环境。函数定义了初始化 ODBC 函数、连接函数、执行 SQL 语句函数以及查询客户名称函数、插入客户名称函数和删除客户名称函数。在使用 ODBC API 时，需要引入 sqlext.h 文件，其中定义了 ODBC API 的函数和结构等信息。MyODBCAPI 类的实现文件如下：

```
01  MyODBCAPI::MyODBCAPI()             //MyODBCAPI 类的构造函数
02  {
03      InitODBC();                    //初始化 ODBC 数据源
```

```
04  }
05  MyODBCAPI::~MyODBCAPI()          //MyODBCAPI 类的析构函数
06  {
07      //如果语句句柄不为 NULL，则释放语句句柄
08      if (hstmt != NULL)
09          SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
10      if (hdbc != NULL)            //如果 ODBC 数据源句柄不为 NULL
11      {
12          SQLDisconnect(hdbc);     //断开 ODBC 数据源连接
13          //释放 ODBC 句柄
14          SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
15      }
16      //释放环境句柄
17      if (henv != NULL)
18          SQLFreeHandle(SQL_HANDLE_ENV, henv);
19  }
```

上面两个函数定义分别是 MyODBCAPI 类的构造函数和析构函数，在构造函数中会执行初始化 ODBC 环境的处理，在析构函数中会依次释放语句句柄、连接句柄和环境句柄。ODBC 初始化函数定义的代码如下：

```
01  BOOL MyODBCAPI::InitODBC()       //初始化 ODBC
02  {
03      henv = NULL;                 //初始化环境句柄为 NULL
04      hdbc = NULL;                 //初始化连接句柄为 NULL
05      hstmt = NULL;                //初始化语句句柄为 NULL
06      bInit = false;               //初始化返回值为 false
07      //分配环境句柄
08      retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
09      //如果成功
10      if ((retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO))
11      {
12          //设置 ODBC 版本属性，如果成功，则设置函数返回值为 true
13          retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
14                                  (void*)SQL_OV_ODBC3,0);
15          if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
16              bInit = true;
17      }
18      return bInit;                //返回初始化结果
19  }
```

上面代码首先将各个句柄初始化为 NULL，然后使用 SQLAllocHandle()函数创建 ODBC 环境句柄。如果成功，则调用 SQLSetEnvAttr()方法设置 SQL_ATTR_ODBC_VERSION 属性为 3.0，即设置 ODBC 版本为 ODBC 3.0。如果设置成功，则为 bInit 变量赋值为 true，表示 ODBC API 环境已经初始化完成。下面代码是连接数据库的函数。

```
01  //连接数据库
02  BOOL MyODBCAPI::Connect(CString odbcName,
03                          CString userID, CString pass)
04  {
05      if (!bInit)      //如果 SQL 环境句柄初始化失败，则赋值错误消息，并返回
06      {
07          msg = "初始化 ODBC API 失败";
08          return false;
09      }
10      retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
```

```
11      //分配连接句柄
12      //如果连接成功
13      if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
14      {
15          //设置数据库连接的超时时间为 10 秒
16          SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (void*)10, 0);
17          //设置超时时间为 10 秒
18          //连接数据源
19          retcode = SQLConnect(hdbc, (SQLCHAR*)(LPCTSTR)odbcName,
20                                  SQL_NTS,(SQLCHAR*)(LPCTSTR)userID,
21                              SQL_NTS,(SQLCHAR*)(LPCTSTR)pass,SQL_NTS);
22          //如果成功
23          if (retcode ==SQL_SUCCESS ||retcode == SQL_SUCCESS_WITH_INFO)
24          {
25              //设置成功消息，并返回 true
26              msg = "连接数据源成功";
27              return true;
28          }
29          else          //如果连接失败
30          {
31              //设置失败消息，并返回 false
32              msg = "连接数据源失败";
33              return false;
34          }
35      }
36      else     //如果分配连接句柄失败，则设置失败消息，并返回 false
37      {
38          msg = "分配连接句柄失败";
39          return false;
40      }
41  }
```

上面代码显示了如何连接数据源。首先判断 ODBC 环境句柄是否已经初始化，如果成功初始化，则调用 SQLAllocHandle()函数分配连接句柄，分配成功后，调用 SQLSetConnectAttr()函数设置连接的超时时间为 10 秒。接着使用传入的参数调用 SQLConnect()函数连接指定的 ODBC 数据源，并返回结果。下面的 ExecSQL()函数显示了如何使用 ODBC API 执行 SQL 语句。

```
01  //执行 SQL 语句
02  BOOL MyODBCAPI::ExecSQL(CString sql)
03  {
04      retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
05      //分配语句句柄
06      //如果成功
07      if(retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
08      {
09          //如果分配语句句柄成功，则执行相应的 SQL 语句
10          retcode = SQLExecDirect(hstmt, (SQLTCHAR*)(LPCTSTR)sql, sql.
11                                      GetLength());
12          if(retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
13          {
14              //如果执行 SQL 语句成功，则设置成功消息，并返回 true
15              msg = "执行 SQL 语句成功";
16              return true;
17          }
18          else
```

```
19          {
20              //如果执行 SQL 语句失败，则设置失败消息，并返回 false
21              msg = "执行 SQL 语句失败";
22              return false;
23          }
24      }
25      else //否则，设置失败消息，并返回 false
26      {
27          msg = "分配语句句柄失败";
28          return false;
29      }
30  }
```

上面代码调用 SQLAllocHandle()函数分配语句句柄后，调用 SQLExecDirect()函数执行 SQL 查询，并返回操作结果。下面是使用此函数执行查询客户名称和联系人姓名的代码。

```
01  //查询所有客户的公司名称和联系人姓名
02  void MyODBCAPI::QueryCustomer()
03  {
04      if (!Connect("Test", "sa", "sa"))   //连接 Test 数据源
05      {
06          //如果成功，则设置成功消息，并返回
07          msg = "连接数据源失败\n";
08          return;
09      }
10      //执行 SQL 语句
11      if (!ExecSQL("SELECT CompanyName, ContactName FROM Customers"))
12      {
13          //如果执行查询客户公司名和联系人姓名语句失败，则设置失败消息，并返回
14          msg = "执行 SQL 语句失败\n";
15          return;
16      }
17      //结果信息
18      msg = "查询到的客户信息如下\r\n 编号\t 公司名称\t 联系人姓名\r\n";
19      //存放公司名称和联系人名称的变量
20      CString CompanyName, ContactName;
21      long cbNameLen = 500;        //名称字段长度为 500
22      //绑定公司名称字段
23      SQLBindCol(hstmt, 1, SQL_C_CHAR,
24          (void*)(LPCTSTR)CompanyName.GetBuffer(cbNameLen), cbNameLen,
25          &cbNameLen);
26      //绑定联系人名称字段
27      SQLBindCol(hstmt, 2, SQL_C_CHAR,
28          (void*)(LPCTSTR)ContactName.GetBuffer(cbNameLen), cbNameLen,
29          &cbNameLen);
30      int i = 0;                   //记录计数变量初始化为 0
31      while (SQLFetch(hstmt) == SQL_SUCCESS)  //移动记录
32      {
33          i++;                     //记录计数变量增加 1
34          if (retcode == SQL_NO_DATA_FOUND)
35              break;
36          //如果没有找到数据，则退出 while 循环
37          CString info;            //记录信息变量
38          info.Format("%d%s%s\r\n", i, CompanyName, ContactName);
39          //格式化记录信息
40          msg += info;             //将记录信息增加到结果信息中
41      }
42  }
```

上面代码显示了查询公司名称和联系人姓名的代码，调用 Connect()函数和 ExecSQL()函数执行查询语句。执行成功后，通过 SQLBindCol()函数绑定要获取的列信息到程序变量中，调用 SQLFetch()函数检索记录行，并将返回的记录数据存储到 CString 类型的变量中，

其他程序可以通过 MyODBCAPI 类的 msg 成员变量访问这些结果值。下面的代码是演示如何执行插入客户名称和删除客户名称的功能。

```
01  //插入客户信息
02  void MyODBCAPI::InsertCustomer(CString CustomerID,
03                                 CString CompanyName)
04  {
05      if (!Connect("Test", "sa", "sa"))       //连接 Test 数据源
06      {
07          //如果成功，则设置成功消息，并返回
08          msg = "连接数据源失败\n";
09          return;
10      }
11      CString sql;                            //SQL 语句变量
12      //格式化要执行的插入 SQL 语句
13      sql.Format("INSERT into  Customers (CustomerID,
14           CompanyName)values('%s', '%s')",    CustomerID, CompanyName);
15      if (!ExecSQL(sql))                      //执行 SQL 语句
16      {
17          //如果执行 SQL 语句失败，则设置错误信息，并返回
18          msg = "插入客户信息失败\n";
19          return;
20      }
21      msg = "插入客户信息成功.SQL=" + sql;       //设置成功提示信息
22      return;                                 //函数返回
23  }
24  //删除客户信息
25  void MyODBCAPI::DeleteCustomer(CString CustomerID)
26  {
27      if (!Connect("Test", "sa", "sa"))       //连接 Test 数据源
28      {
29          //如果成功，则设置成功消息，并返回
30          msg = "连接数据源失败\n";
31          return;
32      }
33      CString sql;                            //SQL 语句变量
34      //格式化要执行的删除 SQL 语句
35      sql.Format("DELETE FROM Customers WHERE CustomerID='%s'",
36                  CustomerID);
37      if (!ExecSQL(sql))                      //执行 SQL 语句
38      {
39          //如果执行 SQL 语句失败，则设置错误信息，并返回
40          msg = "删除客户信息失败\n";
41          return;
42      }
43      msg = "删除客户信息成功.SQL=" + sql;       //设置成功提示信息
44      return;                                 //函数返回
45  }
```

上面代码分别调用 Connect()函数和 ExecSQL()函数执行插入客户名称和删除客户名称的工作。编写好 ODBC API 封装类后，在程序中就可以调用此类执行这些函数，代码如下：

```
01  //连接 ODBC 数据源
02  void CODBCAPISampleView::OnMenuitemConnect()
03  {
04       MyODBCAPI myODBC;                 //定义自定义类 MyODBCAPI 的变量
```

```
05        myODBC.QueryCustomer();      //调用 QueryCustomer()方法查询客户信息
06        //在日志框中显示查询到的数据
07        m_editLog.SetWindowText(myODBC.msg);
08    }
```

上面代码显示了在 MFC 程序中，单击“数据库”|“连接数据库”命令时执行的操作，会调用 MyODBCAPI 类的 QueryCustomer()函数查询所有的客户公司名称和联系人姓名，并在对话框的日志编辑框中显示出来。查询客户公司的运行效果如图 13-9 所示。

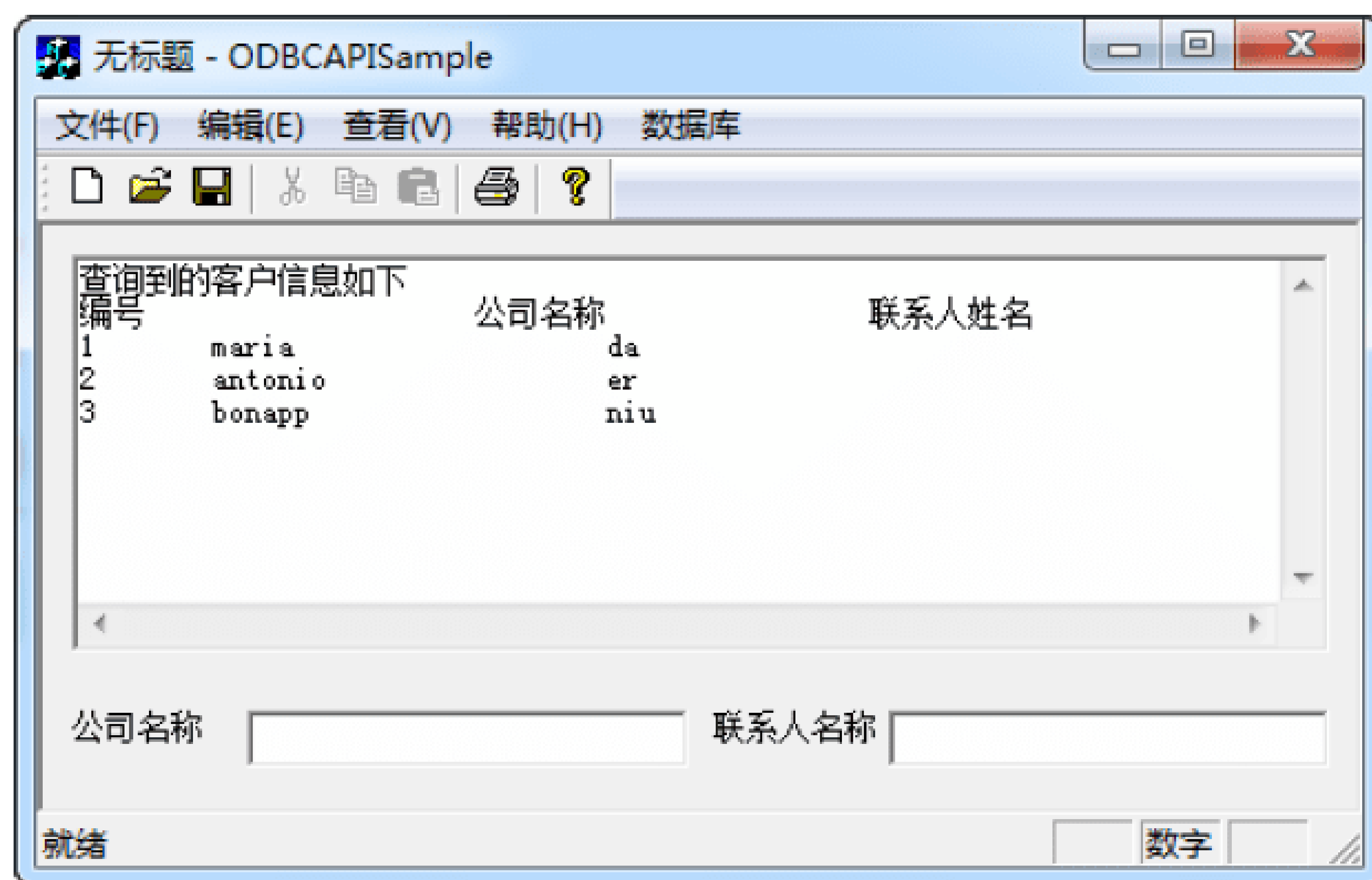


图 13-9　查询客户公司的运行效果

下面代码显示了在 MFC 程序中单击“数据库”|“添加客户”命令时执行的操作。

```
01  void CODBCAPISampleView::OnMenuitemInsertcustomer()//增加客户
02  {
03      UpdateData(true);                       //同步控件变量和控件输入值
04      MyODBCAPI myODBC;                       //定义 MyODBCAPI 类的变量
05      //调用 InsertCustomer()方法插入
06      myODBC.InsertCustomer(m_company, m_name);
07      m_editLog.SetWindowText(myODBC.msg);//在日志框中显示操作结果
08  }
```

调用 MyODBCAPI 类的 InsertCustomer()函数，将客户编号编辑框中的信息和公司名称中的信息插入到数据库 Customers 表中。增加客户公司的运行效果如图 13-10 所示。

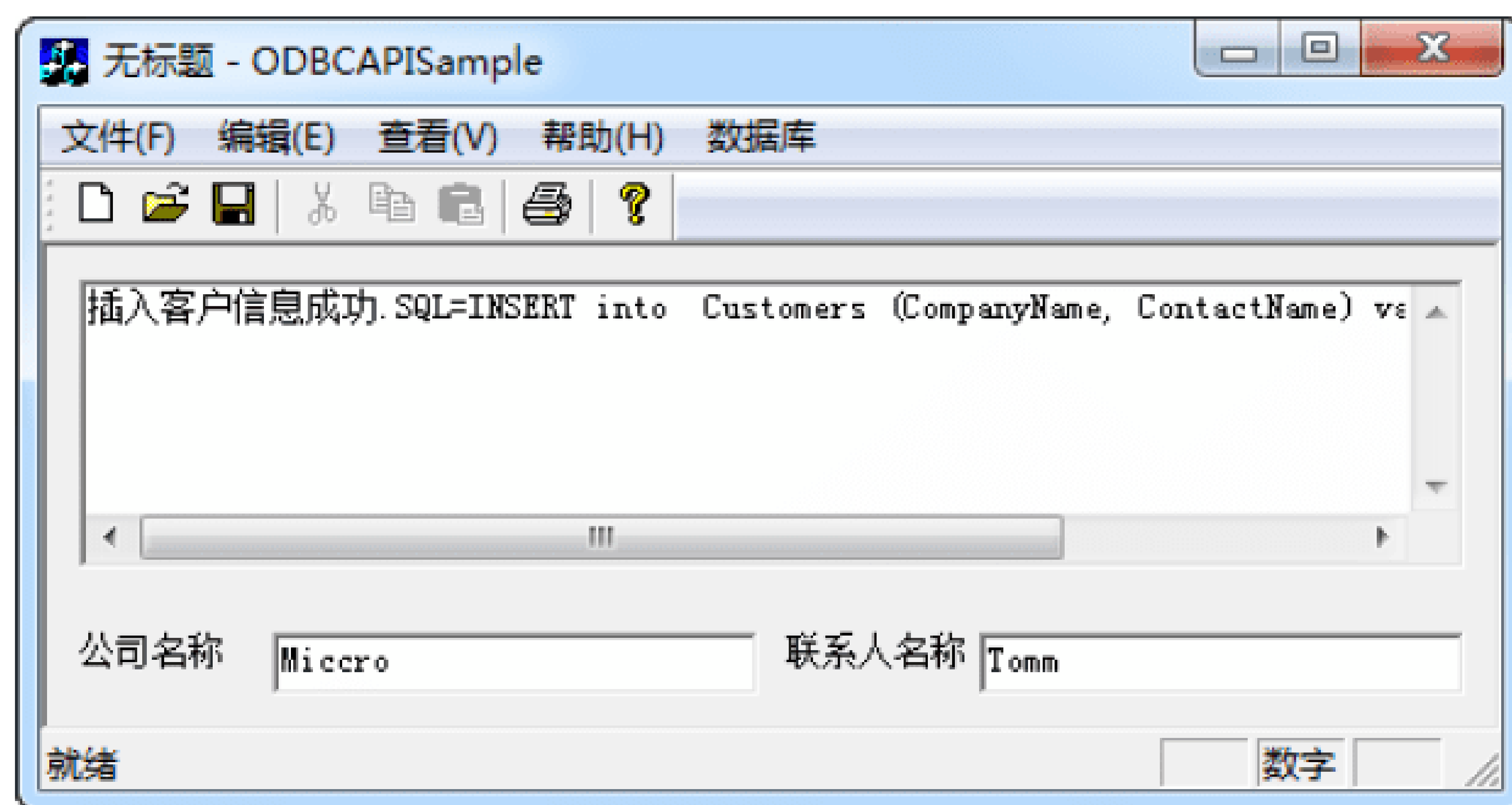


图 13-10　增加客户公司的运行效果

下面代码显示了在 MFC 程序中单击“数据库”|“删除客户”命令时执行的操作。

```
01  void CODBCAPISampleView::OnMenuitemDeletecustomer()//删除客户
02  {
```

```
03      UpdateData(true);                      //同步控件变量和控件输入值
04      MyODBCAPI myODBC;                      //定义 MyODBCAPI 类的变量
05      myODBC.DeleteCustomer(m_company);      //调用 DeleteCustomer()方法删除
06      m_editLog.SetWindowText(myODBC.msg);//在日志框中显示操作结果
07  }
```

调用 MyODBCAPI 类的 DeleteCustomer()函数，将客户编号编辑框中的客户信息从数据库 Customers 表中删除，删除客户公司的运行效果如图 13-11 所示。

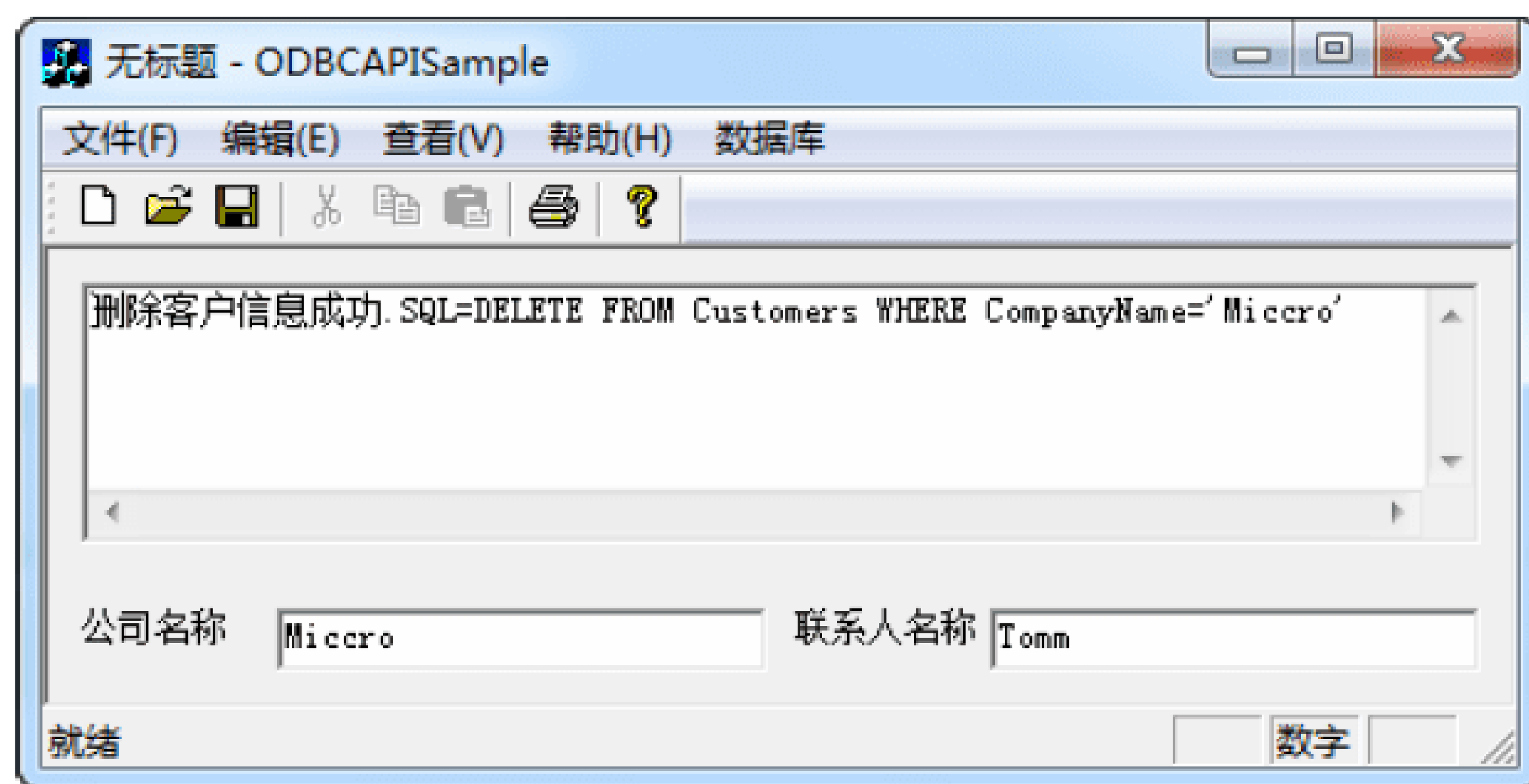


图 13-11 删除客户公司的运行效果

13.3 用 MFC ODBC 类操作数据库

前两节中介绍了 ODBC API 的编程知识和操作实例，从中可以看出，使用 ODBC API 的过程很繁琐。为了提高编程效率，MFC 对 ODBC API 进行了封装，提供了一组 MFC ODBC 类来操作 ODBC 数据源。本节将介绍基于 ODBC 数据库类的 MFC 类及其使用。

13.3.1 连接数据库——CDatabase 类

MFC ODBC 类结合 ODBC API 封装了一组操作 ODBC 数据源必须的 MFC 类，核心实现是对 ODBC API 的封装。换言之，MFC ODBC 类简化了 ODBC API 的调用。虽然数据库类封装了 ODBC 的功能，但是与 ODBC API 函数不是一对一的。数据库类提供了更高级别的抽象。

要通过 MFC ODBC 访问数据源，首先需要通过 CDatabase 类建立到数据源的连接。当使用完数据连接时，应该关闭 CDatabase 对象，并销毁或重利用到新连接上。在同一个程序中，可以同时使用多个 CDatabase 对象。

在 ODBC 管理器中配置完 ODBC 数据源后，就可以连接到指定的 ODBC 数据源。首先构造 CDatabase 对象，然后调用 CDatabase 对象的 OpenEx()或 Open()成员函数打开到数据源的连接。具体代码如下：

```
01  CDatabase m_dbCust;                            //定义 CDatabase 类变量
02  m_dbCust.OpenEx( _T( "DSN=MYDB;UID=LLN " ),
03                 CDatabase::openReadOnly | CDatabase::noOdbcDialog );
04                                                  //打开到 ODBC 数据源的连接
```

表 13-2 中列出了 CDatabase 类的常用成员。

表 13-2　CDatabase类的常用成员

成　　员	含　　义
m_hdbc()	ODBC数据源的连接句柄，类型为HDBC
CDatabase()	构造CDatabase对象。要初始化对象还必须调用OpenEx()函数或Open()函数
Open()	通过ODBC驱动，建立到数据源的连接
OpenEx()	通过ODBC驱动，建立到数据源的连接
Close()	关闭数据源连接
GetConnect()	返回ODBC连接字符串，是CDatabase类连接到数据源时使用的连接串
IsOpen()	返回当前CDatabase对象是否已经连接到数据源。如果连接，则返回true
GetDatabaseName()	返回当前使用的数据库的名称
CanUpdate()	返回CDatabase对象是否可编辑，即是否是只读的。如果可编辑，则返回true
CanTransact()	返回数据源是否支持事务。如果数据源支持事务，则返回true
SetLoginTimeout()	设置连接数据库的超时时间，单位是秒
SetQueryTimeout()	设置数据查询操作的超时时间。会影响后续数据库操作的超时时间
BeginTrans()	启动事务，其可以使得后续的AddNew、Edit、Delete和Update等操作可逆，需要数据源支持事务，才有效
BindParameters()	在调用CDatabase::ExecuteSQL之前，绑定参数
CommitTrans()	提交事务，提交BeginTrans后的所有操作
Rollback()	回滚事务，放弃BeginTrans后的所有操作，使数据库恢复BeginTrans之前的状态
Cancel()	从另一个线程中取消异步操作
ExecuteSQL()	执行SQL语句，不返回数据记录

13.3.2　选择和操作记录——CRecordset 类

一旦 CDatabase 对象连接到了数据源，就可以使用记录类 CRecordset 查询或选择记录集，或者直接执行 SQL 语句完成事务或事务回滚。记录类可以是从 CRecordset 类继承而来的与特定应用相关的类。当定义一个 CRecordset 类时，需要指定与之关联的数据源、使用的表和列。使用类向导或应用向导可以创建与指定数据源相连的记录集，向导使用 CRecordset 类的 GetDefaultSQL()函数返回表名。使用 CRecordset 对象，可以完成如下操作。

- 检验当前记录的数据域。
- 过滤或排序记录集。
- 定制默认的 SELECT 语句。
- 在选择的记录中导航。
- 如果数据源和记录集都是可编辑的，可以增加、修改或删除记录。
- 测试记录集是否允许重查询，并刷新记录集内容。

当使用完 CRecordset 对象后，应该关闭并销毁它。当关闭 CDatabase 对象时，会自动关闭与 CDatabase 对象相连的所有 CRecordset 对象。如下方法是创建记录集类 CRecordset 的步骤。

（1）使用前面介绍过的方法，打开“添加类”对话框，如图 13-12 所示。

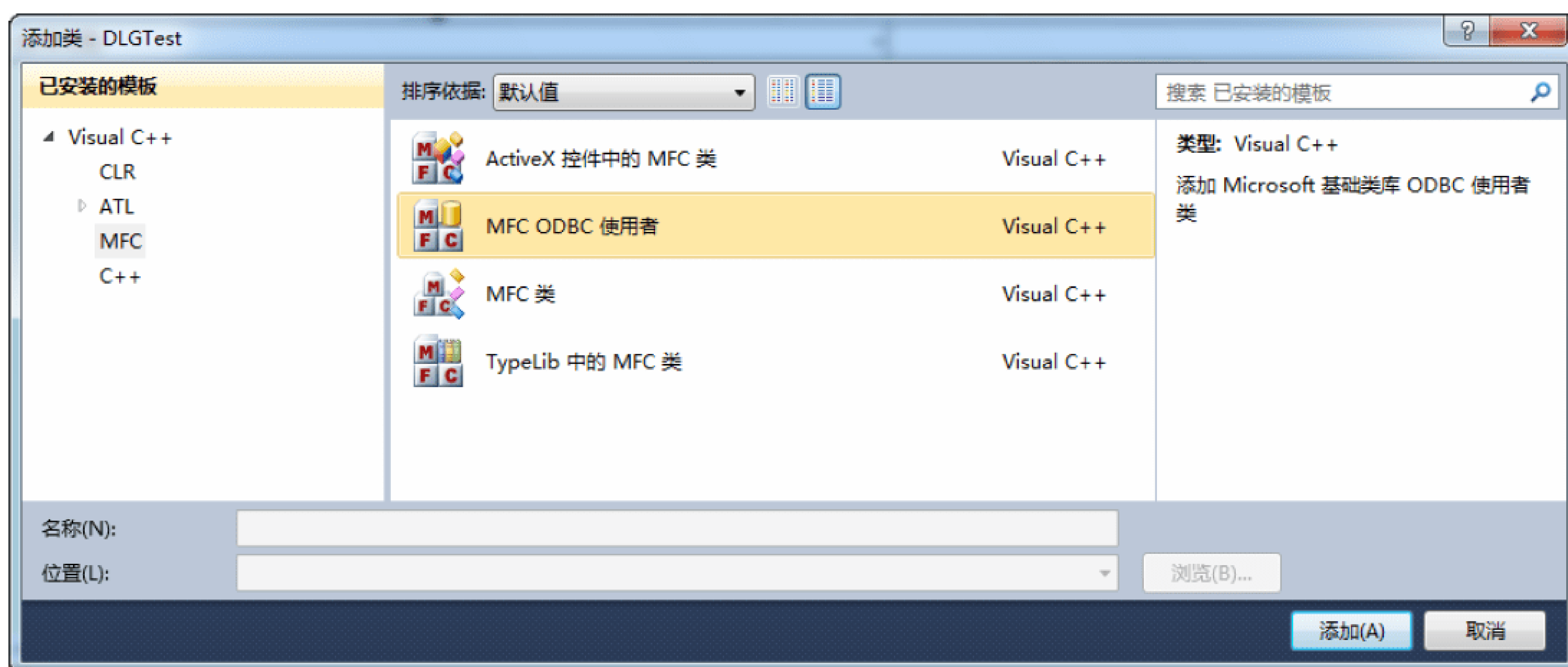
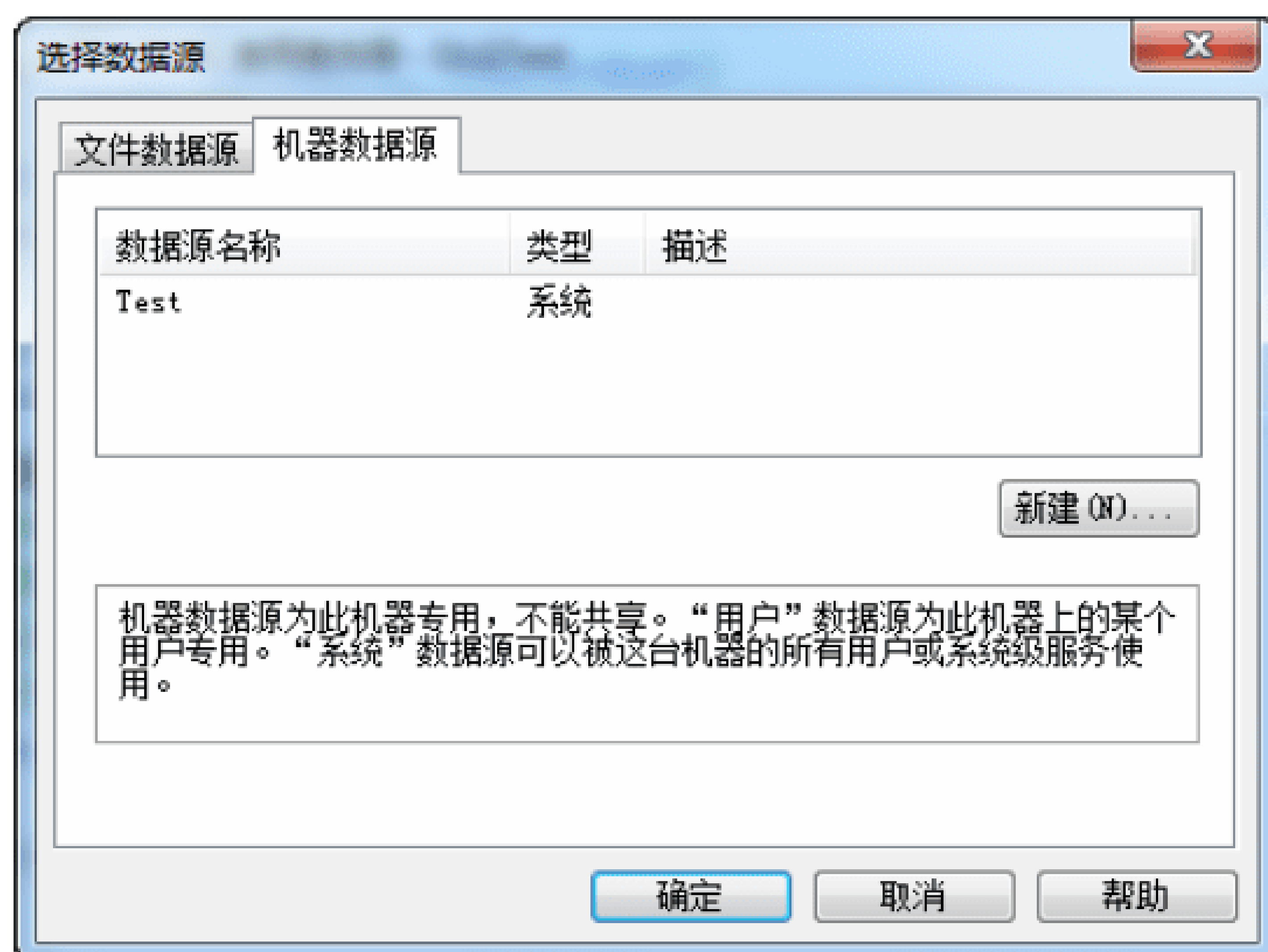


图 13-12　“添加类”对话框

（2）选择“MFC ODBC 使用者”列表项，单击“添加”按钮，弹出“MFC ODBC 使用者向导”对话框，如图 13-13 所示。单击“数据源”按钮，弹出 “选择数据源”对话框，如图 13-14 所示。选择数据源，单击“确定”按钮，弹出“选择数据库对象”对话框，如图 13-15 所示。选择表和视图对象，单击“确定”按钮返回，即添加了继承自 CRecordset 的类。



图 13-13　“MFC ODBC 使用者向导”对话框



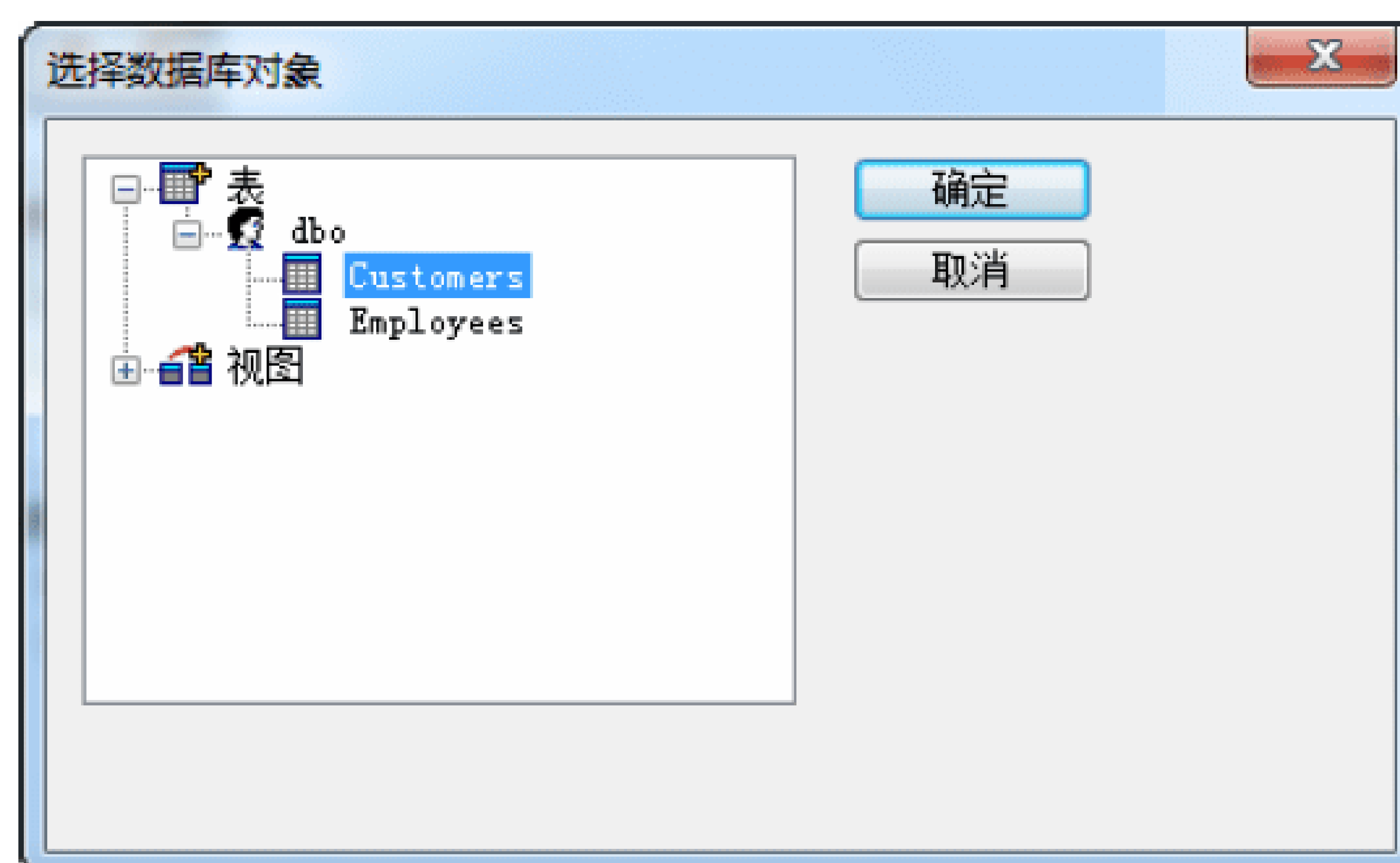


图 13-14　“选择数据源”对话框　　　图 13-15　“选择数据库对象”对话框

13.3.3　在窗体中显示和操作数据——CRecordView 类

有些数据访问应用程序选择数据，并在窗体中显示。在 MFC 中，提供了继承自 CFormView 类的 CRecrodView 类，可以直接连接到 CRecordset 对象。使用 DDX（对话框数据交换）根据窗体中的记录集控件的当前记录，显示每个域的值，并可以将更新的信息更新回记录集。然后 CRecordset 使用 RFX（记录集域交换）在域数据成员和数据源中相应表之间进行同步。使用应用向导或类向导可以创建记录视图类和与其关联的记录类。当关闭文档时，记录视图和记录会被销毁。使用 CRecordView 类的 OnGetRecordset()函数可以获取指向当前记录的对象的指针，其函数原型为：

```
virtual CRecordset* OnGetRecordset( ) = 0;          //获取记录集的虚函数
```

其返回值为 CRecordset 类型，可以将其转换为自定义的 CRecordset 类型的对象。通过处理 CRecordView 类对象的 OnMove 事件，当当前记录修改或定位记录时，读者可以根据自己的需求进行处理，如在界面上显示提示信息。使用 IsOnFirstRecord()函数和 IsOnLastRecord()函数可以判断当前记录是否是记录集中的第一条记录和最后一条记录。

13.3.4　异常处理——CDBException 类

MFC 中使用 CDBException 类处理从数据库类中抛出的异常。此类包括两个公共数据成员，可以确定异常原因和描述异常的文本消息。CDBException 对象由数据库类的成员函数构造并抛出。异常是导致程序不正常运行的情况，如程序控制错误、数据源连接错误或 I/O 处理等。读者可以在 CATCH 语句的范围内捕获异常，也可以使用 AfxThrowDBException()全局函数获取异常错误。数据成员包括以下 3 个。

- m_nRetCode 数据成员：包含 ODBC 返回的错误代码，类型为 RETCODE。
- m_strError 数据成员：包含描述错误的字符串。
- m_strStateNativeOrigin 数据成员：包含 ODBC 返回的错误代码对应的描述字符串。

读者可以在代码中使用这 3 个数据成员进行错误处理，代码如下：

```
01  Try
02  {
03      //执行 MFC ODBC 操作
04  }
05  CATCH(CDBException ee)
06  {
07      //此处调用 ee 对象的数据成员，提示用户在执行数据库操作时发生的错误
08      MessageBox(ee.m strError);
09  }
```

以上代码在 CATCH 语句块中，使用 CDBException 类捕获发生的错误，并可以根据需要处理错误信息。

13.3.5　断开数据源连接

在调用 CDatabase 对象的 Close()成员方法之前，必须关闭任何打开的记录集。与要关

闭的 CDatabase 对象相连的记录和未完成的 AddNew 或 Edit 语句会被取消，并且所有未完成的事务都会回滚。步骤如下。

（1）调用 CDatabase 对象的 Close()成员函数。

（2）销毁 CDatabase 对象，如果需要重用数据源连接，则再次调用 CDatabase 对象的 OpenEx()或 Open()成员函数。

具体代码如下：

```
01  m_dbCust.Close( );          //关闭数据源连接
02  m_dbCust.OpenEx("DSN=MYDB;UID=LLN ");
03                              //调用 OpenEx()函数重新打开到数据源的连接
```

13.3.6 MFC ODBC 操作数据库实例

前面几小节讲解了如何使用 MFC ODBC 类操作数据库，本小节以操作 TryAgain 数据库中的 Customer 表为例，讲解如何使用 MFC ODBC 操作数据库。按照前面讲述的方法创建记录集 CRSCustomer 类和记录视图 CRVCustomer 类。在 MFC 主程序的应用程序类的 InitInstance()函数中，将单文档框架中的视图类修改为 CRVCustomer 类，代码如下：

```
01  BOOL CMFCODBCSampleApp::InitInstance()
02  {
03      ...
04      CSingleDocTemplate* pDocTemplate;       //定义单文档模板变量
05      pDocTemplate = new CSingleDocTemplate(  //实例化单文档模板变量
06          IDR_MAINFRAME,                      //单文档模板对应的资源 ID
07          RUNTIME_CLASS(CMFCODBCSampleDoc),   //对应的文档类
08          RUNTIME_CLASS(CMainFrame),          //对应的主窗口类
09          RUNTIME_CLASS(CRVCustomer);         //对应的客户记录视图类
10      AddDocTemplate(pDocTemplate);           //增加单文档模板
11      ...
12  }
```

在 CRVCustomer 类中填充记录操作函数，如下代码所示：

```
01  CRVCustomer::CRVCustomer() : CRecordView(CRVCustomer::IDD)
02                                      //客户信息记录视图类构造
03  {
04      //{{AFX_DATA_INIT(CRVCustomer)      //数据初始化开始
05      m_pSet = NULL;                      //初始化记录集变量为 NULL
06      //}}AFX_DATA_INIT                   //数据初始化结束
07  }
08  CRVCustomer::~CRVCustomer()             //客户信息记录视图类析构函数
09  {
10      if (m_pSet) delete m_pSet;          //如果记录集变量有效，则删除释放
11  }
```

上面代码显示了客户视图类的构造函数和析构函数，在构造函数中，初始化记录集变量为 NULL，在析构函数中释放记录集对象。下面代码显示了记录视图类如何与记录集成员变量相关联。

```
01  //数据交换函数
02  void CRVCustomer::DoDataExchange(CDataExchange* pDX)
03  {
04      CRecordView::DoDataExchange(pDX);//基类记录集视图的数据交换函数
05      //{{AFX_DATA_MAP(CRVCustomer)    //开始数据映射
06      //将 CustomerID 字段映射到 IDC_EDIT_CUSTOMERID 控件
07      DDX FieldText(pDX, IDC EDIT CUSTOMERID,
08          m_pSet->m_CustomerID, m_pSet);
09      //将 Address 字段映射到 IDC_EDIT_ ADDRESS 控件
10      DDX FieldText(pDX, IDC EDIT ADDRESS,
11          m pSet->m Address, m pSet);
12      //将 City 字段映射到 IDC_EDIT_ CITY 控件
13      DDX FieldText(pDX, IDC EDIT CITY,
14          m pSet->m City, m pSet);
15      //将 CompanyName 字段映射到 IDC_EDIT_ COMPANYNAME 控件
16      DDX FieldText(pDX, IDC EDIT COMPANYNAME,
17          m pSet->m CompanyName, m pSet);
18      //将 ContactName 字段映射到 IDC_EDIT_ CONTACTNAME 控件
19      DDX FieldText(pDX, IDC EDIT CONTACTNAME,
20          m pSet->m ContactName, m pSet);
21      //将 ContactTitle 字段映射到 IDC_EDIT_ CONTACTTITLE 控件
22      DDX_FieldText(pDX, IDC_EDIT_CONTACTTITLE,
23          m pSet->m ContactTitle, m pSet);
24      //将 Country 字段映射到 IDC EDIT  COUNTRY 控件
25      DDX_FieldText(pDX, IDC_EDIT_COUNTRY,
26          m pSet->m Country, m pSet);
27      //将 Fax 字段映射到 IDC EDIT  FAX 控件
28      DDX_FieldText(pDX, IDC_EDIT_FAX,
29          m_pSet->m_Fax, m_pSet);
30      //将 PostalCode 字段映射到 IDC EDIT  POSTALCODE 控件
31      DDX FieldText(pDX, IDC EDIT POSTALCODE,
32          m_pSet->m_PostalCode, m_pSet);
33      //将 Region 字段映射到 IDC_EDIT_ REGION 控件
34      DDX FieldText(pDX, IDC EDIT REGION,
35          m_pSet->m_Region, m_pSet);
36      //将 Phone 字段映射到 IDC_EDIT_ TELE 控件
37      DDX FieldText(pDX, IDC EDIT TELE,
38          m_pSet->m_Phone, m_pSet);
39      //}}AFX_DATA_MAP                //结束数据映射
40  }
```

从上面的代码中可以看出，在记录集视图类中，将编辑框与记录对象的成员变量相关联，这样可以将界面操作与实际的记录操作的数据对应起来。下面代码是获取记录集和记录的实现函数。

```
01  CRecordset* CRVCustomer::OnGetRecordset()  //获取记录集
02  {
03      if (m_pSet != NULL)
04          return m pSet;
05      //如果记录集不为 NULL，则返回当前记录集
06      m_pSet = new CRSCustomer(NULL);     //创建 CRSCustomer 类型的记录集
07      m_pSet->Open();                         //打开记录集
08      return m pSet;                          //返回打开的记录集
09  }
10  CRSCustomer* CRVCustomer::GetRecordset()   //获取记录
```

```
11  {
12      CRSCustomer* pData = (CRSCustomer*) OnGetRecordset();
13      //调用 OnGetRecordset()方法
14      //判断创建的记录变量是否为 CRSCustomer 类型
15      ASSERT(pData == NULL
16          || pData->IsKindOf(RUNTIME_CLASS(CRSCustomer)));
17      return pData;                           //返回获取的记录
18  }
```

上面代码初始化记录视图中的记录集为 CRSCustomer 类，并打开到数据库的连接获取数据。

```
01  void CRVCustomer::OnInitialUpdate()         //初始化更新
02  {
03      BeginWaitCursor();                      //将光标变为等待光标
04      GetRecordset();                         //获取客户记录集
05      CRecordView::OnInitialUpdate();         //调用基类的初始化更新函数
06      if (m_pSet->IsOpen())                   //判断记录集是否已经打开
07      {
08          //如果已经打开
09          CString strTitle = m_pSet->m_pDatabase->GetDatabaseName();
10          //获取数据库名称
11          CString strTable = m_pSet->GetTableName();//获取表名
12          if (!strTable.IsEmpty())
13              strTitle += _T(":") + strTable;
14          //组合文档标题
15          GetDocument()->SetTitle(strTitle);
16          //设置文档的标题为数据库名+表名
17      }
18      EndWaitCursor();                        //恢复光标
19  }
```

上面代码会在初始化对话框的函数中获取记录集数据，并将数据库名称作为标题显示在程序的标题栏上。

```
01  void CRVCustomer::OnMenuitemAddrecord()//增加记录
02  {
03      if (m_pSet->IsOpen())               //判断记录集是否已经打开
04      {
05          //如果打开记录集
06          m_pSet->AddNew();               //调用 AddNew()函数增加记录集
07          UpdateData(false);              //并使用记录初始值初始化字段控件
08      }
09  }
10  void CRVCustomer::OnMenuitemDeleterecord()//删除记录
11  {
12      if (m_pSet->IsOpen())
13          m_pSet->Delete();
14      //如果记录集已经打开，则删除当前记录
15  }
16  void CRVCustomer::OnMenuitemUpdaterecord()//修改记录
17  {
18      if (m_pSet->IsOpen())
19          m_pSet->Edit();
```

```
20      //如果记录集已经打开，则开始修改当前记录
21  }
22  void CRVCustomer::OnMenuitemCommit()        //增加或修改提交
23  {
24      if (m pSet->IsOpen())                   //判断记录集是否已经打开
25      {
26          //如果打开记录集
27          UpdateData(true);                   //使用控件数据更新记录字段值
28          m_pSet->Update();                   //更新记录集
29          m_pSet->MoveFirst();                //移动到记录集的第一条记录
30          UpdateData(false);          //使用当前记录的字段值更新字段控件
31      }
32  }
33  void CRVCustomer::OnMenuitemRefresh()       //刷新记录
34  {
35      if (m pSet != NULL)
36          m pSet->Requery();
37      //如果记录集不为 NULL，则重新查询记录集
38      else                                    //否则
39      {
40          m pSet = new CRSCustomer(NULL);     //实例化 CRSCustomer 记录集
41          m_pSet->Open();                     //调用 Open()函数打开记录集
42      }
43  }
44  void CRVCustomer::OnMenuitemCancelrecord()  //取消修改记录
45  {
46      if (m pSet->IsOpen())                   //如果记录集已经打开
47      {
48          m pSet->CancelUpdate();             //取消最近的更新
49          m_pSet->MoveFirst();                //移动到记录集的第一条记录
50          UpdateData(false);          //使用当前记录的字段值更新字段控件
51      }
52  }
```

上面代码显示了如何实现记录集的添加、删除、修改、更新、取消记录集更新以及刷新记录集等功能的实现。根据注释，可以容易地理解函数的调用。编译运行此程序，运行效果如图 13-16 所示。

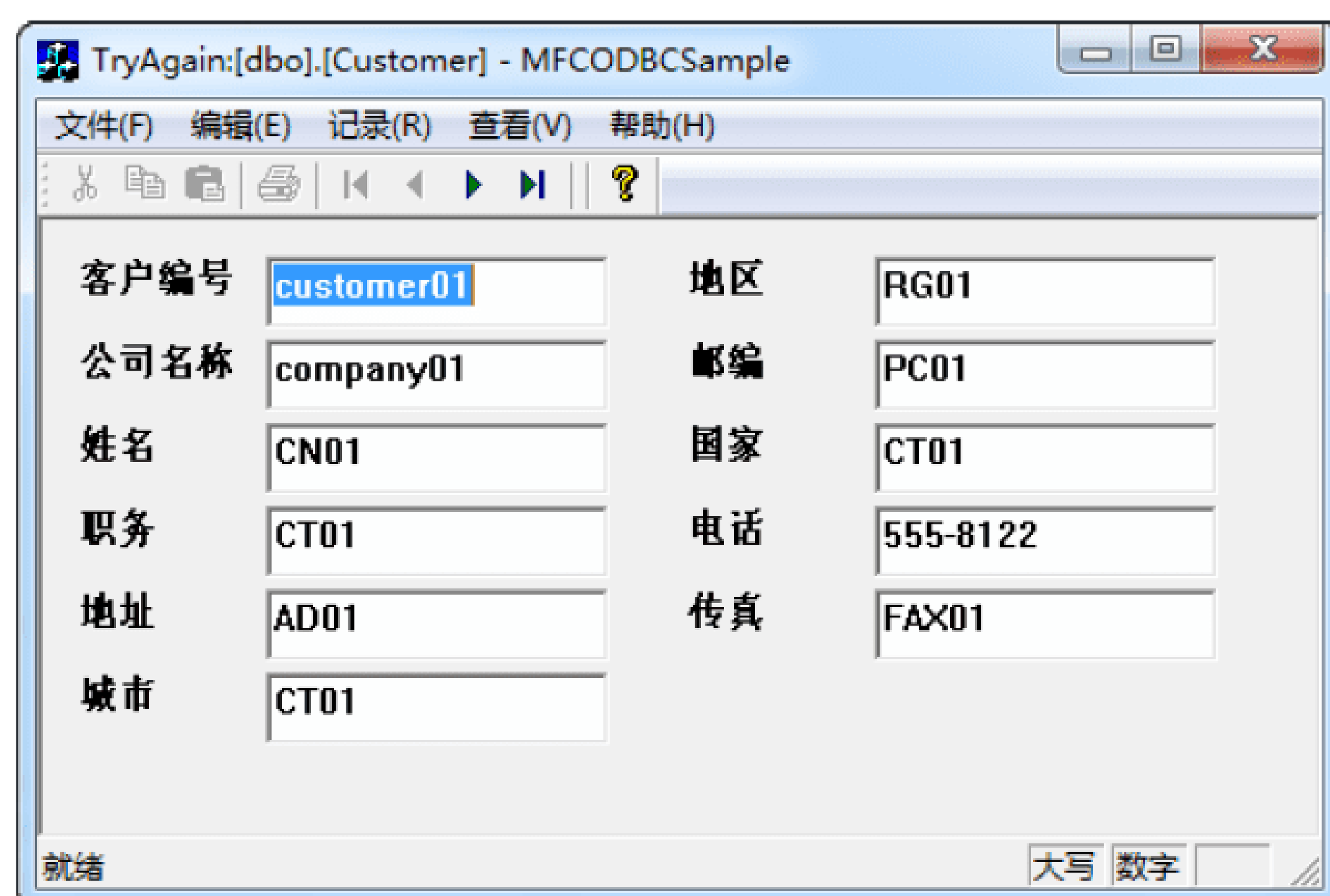


图 13-16　MFC ODBC 操作数据库运行效果

从上面的过程可以看出，使用 MFC ODBC 进行数据库的编程比使用 ODBC API 要简单得多，但是 MFC ODBC 并没有完整地封装 ODBC API 的功能，如需要获取数据库结构和表结构，则必须使用 ODBC API 进行访问。因此，读者需要根据程序的需求确定使用哪种方式访问 ODBC 数据源，既有较高的开发效率又能实现程序的功能。

13.4　自动注册 DSN

ODBC API 组件中提供了 SQLConfigDataSource()函数，可以完成自动注册 DSN 的工作，可以在程序中增加此功能，减少用户安装时的配置工作。此函数可以增加、修改或删除 ODBC 数据源，其函数原型为：

```
BOOL SQLConfigDataSource(
  HWND   hwndParent,                    //指定调用此函数的父对话框的句柄
  WORD   fRequest,                      //指定要执行的操作
  LPCSTR   lpszDriver,                  //指定 ODBC 驱动信息
  LPCSTR   lpszAttributes);             //指定配置的 ODBC 数据源的属性对的集合
```

其中，fRequest 参数用于指定要执行的操作，有效取值如下。

- ODBC_ADD_DSN：增加用户 ODBC 数据源。
- ODBC_CONFIG_DSN：配置用户 ODBC 数据源。
- ODBC_REMOVE_DSN：删除用户 ODBC 数据源。
- ODBC_ADD_SYS_DSN：增加系统 ODBC 数据源。
- ODBC_CONFIG_SYS_DSN：配置系统 ODBC 数据源。
- ODBC_REMOVE_SYS_DSN：删除系统 ODBC 数据源。
- ODBC_REMOVE_DEFAULT_DSN：删除默认 ODBC 数据源。

如果配置 ODBC 数据源成功，则函数返回 true，否则返回 false。在使用此函数时，需要包含 odbcinst.h 头文件，并链接 odbccp32.lib 静态库。以下代码完成自动注册 13.3 节使用的名称为 Test 的 ODBC 数据源。

```
01  void CConfigDSNDlg::OnButtonRegdsn()    //自动注册 DSN
02  {
03      //调用 SQLConfigDataSource()函数注册 DSN 数据源
04      if (SQLConfigDataSource(NULL,ODBC ADD SYS DSN, "SQL Server",
05          "DSN=Test\0" "Server=(local)\0"
06          "Database=TryAgain\0""Trusted Connection=yes\0"))
07          AfxMessageBox("自动注册 DSN 成功");//显示提示成功消息框
08      else
09          AfxMessageBox("自动注册 DSN 失败");//显示提示失败消息框
10  }
```

上面代码注册的 ODBC 数据源的名称为 Test，此数据源连接的是本机上的 SQL Server 服务器，连接的数据库是 TryAgain 数据库，使用信任连接。运行程序，单击“自动注册 DSN”按钮后，会执行此函数自动注册 DSN。程序运行效果如图 13-17 所示。

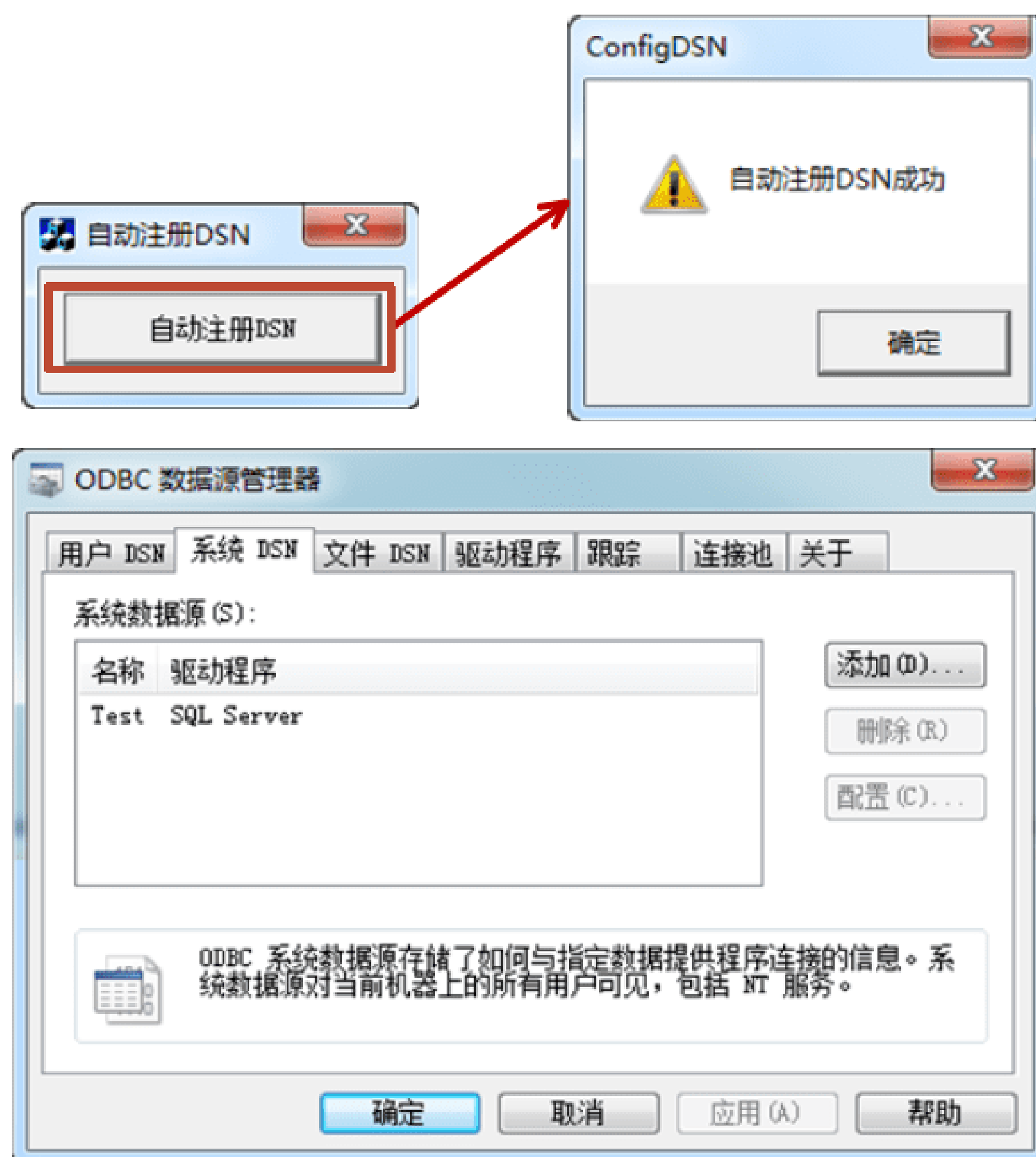


图 13-17　自动注册 DSN 运行效果

13.5　本 章 小 结

本章主要讲述了 ODBC 数据库访问技术。本章重点是掌握 ODBC API 以及 MFC ODBC 的结构以及使用方法。本章的难点是掌握 ODBC API 的知识。第 14 章将介绍在 VC 中访问 OLE DB 的技术。

13.6　习　　题

1．通过“ODBC 数据源管理器”配置一个 ODBC 数据源，命名为 example，连接的数据库是第 12 章中习题创建的数据库 TASK。

【思路】参考 13.1.5 小节所描述的配置步骤来完成。

2．通过 ODBC API 连接第 1 题创建的数据源，然后读取表 January 中的内容并显示在窗口中。

【思路】本题要比 13.2 节的实例完成的功能少得多，只是要求读取表的内容然后显示而已，那么可以有针对性地查看 13.2 节实例相应功能的实现代码即可。

3．通过 MFC ODBC 类来连接数据库 TASK，并读取表 January 中的内容，然后显示在窗口中对应的文本框中（类似图 13-19 所示的界面设计，只是文本框分别对应表中的 ID、name 和 time 字段）。

【思路】可以参考 13.3.6 小节中的实例。

第 14 章　Visual C++中 OLE DB 访问技术

OLE DB 是使用 OLE 组件对象模型，提供访问各种数据源的一组接口。OLE DB 接口提供访问各种不同信息源的数据的接口。这些接口提供 DBMS 对应的数据源，可以共享数据。本章将介绍在 VC 中访问 OLE DB 的技术。

14.1　OLE DB 简介

OLE DB 是为了统一访问各种信息源而定义的一组数据访问接口。使用 OLE DB 可以访问任何具有 OLE DB 驱动程序的数据源，这也是现阶段最常用的数据访问方式。本节主要介绍 OLE DB 的概念，读者通过本节可以从概念上对 OLE DB 有个认识。

14.1.1　什么是 OLE DB

随着信息时代的到来，各行各业开始使用计算机进行行业数据的管理，目前常用的信息管理方式是使用数据库，但是除了使用商业数据库，管理信息还可以使用电子表格、电子邮件等其他方式。这就为信息获取程序的编写带来了复杂度。为了解决这一问题，微软提供了一组用于访问各种数据源的组件接口，即 OLE DB。

从字面上理解，OLE DB 就是数据库链接对象。但实际上，使用 OLE DB 不仅可以访问数据库，还可以访问其他各种类型的数据。要访问一种新格式的数据，只要编写此种格式的 OLE DB 提供程序即可。OLE DB 是一组提供统一的访问存储在各种不同信息源中的数据的 ActiveX 接口。OLE DB 由以下几个 COM 组件组成。

- 枚举器对象（Enumerators）：用于查找可用的数据源和其他枚举器。OLE DB 程序可以使用枚举器查找可以使用的数据源。
- 数据源对象（Datasource）：包含到数据源连接的对象，如电子表格文件或 DBMS，是用于存储会话的对象。
- 会话对象（Sessions）：提供事务处理的上下文。一个数据源对象可以创建多个会话。会话是用于处理事务、命令和记录的对象。
- 事务对象（Transaction）：其是管理最低级别的嵌套事务的对象，可以提交事务处理，也可以终止事务处理。
- 命令对象（Commands）：是执行文本命令，如 SQL 语句的对象。如果文本命令返回记录集，如一个查询语句，则命令对象是记录集的包容器。一个会话对象可以包含多个命令对象。
- 记录集对象（Rowsets）：使用表格的格式显示数据，通常由会话或命令创建记

录集。

- 错误对象（Errors）：可以由任何 OLE DB 对象创建，其中可以包含错误的详细信息，也可以包含用户定制的错误对象。

从上面这些组件可以看出，OLE DB 的编程模型与 ODBC 和 DAO 等编程模型类似，都是由连接、命令和记录集等这些对象实现。但是，OLE DB 又与这些数据库编程模型有所不同，14.1.2 小节将介绍 OLE DB 和 ODBC 之间的关系。

14.1.2　OLE DB 和 ODBC 之间的关系

从 14.1.1 小节的介绍中，可以看出 OLE DB 和 ODBC 很相似，这两者都是提供统一的数据访问接口的编程模型。但是这两者之间又存在很多不同之处。

- 支持的数据源类型有差别。ODBC 是开放的数据库连接接口，即只能处理数据库信息源的数据。而 OLE DB 是统一的数据访问接口，这里的数据既可以是数据库信息源，也可以是电子表格、电子邮件等其他格式的信息源。因此，对于不是数据库格式的信息源，只能使用 OLE DB 编程模型进行访问。
- ODBC 和 OLE DB 都提供直接访问数据库底层实现的接口，但是 OLE DB 还提供对 ODBC 的封装，即 OLE DB 的 ODBC 提供程序。同时，ADO 技术是对 OLE DB 的进一步封装。

因此，OLE DB 和 ODBC 之间是互相联系，又有区别的。可以通过封装 ODBC 的 OLE DB 组件访问各种数据库，也可以直接通过 ODBC 访问各种数据库，或者通过 OLE DB 的各种不同数据源的驱动程序访问各种数据库。但是对于非数据库格式的信息源，则需要使用 OLE DB 来访问。只有清楚地了解各种数据库编程模型的差异，才能编写出更高效的程序。

14.2　Visual C++中的 OLE DB 类

为了提高开发效率，VC 提供了对 OLE DB 类的封装，简化了 OLE DB 程序的开发。主要对数据源、会话、记录集和表进行封装。本节就分别介绍 CDataSource、CSession、CRowset 和 CTable 这几个封装类。

14.2.1　数据库连接类 CDataSource

ATL 中使用 CDataSource 类封装了 OLE DB 的数据库连接类，表示通过提供程序连接到数据源的连接。一个数据连接可以创建一个或多个数据库会话。在创建会话前，需要先调用 CDataSource 类的 Open()函数，打开到数据库的连接。当使用完数据连接时，要记得使用 Close()函数关闭数据源连接。CDataSource 类的成员函数如表 14-1 所示。

表 14-1　CDataSource类的成员函数

函　数　名	功　　能
Close()	关闭连接

续表

函　数　名	功　　能
GetProperties()	获取连接的当前属性值
GetProperty()	获取连接的单个当前属性值
Open()	建立到数据源的连接
OpenWithPromptFileName()	打开相应的数据源，允许用户选择以前创建的数据链接文件
OpenFromFileName()	使用数据链接文件打开数据源
OpenFromInitializationString()	使用连接字符串打开数据源连接
GetInitializationString()	获取当前打开的数据库连接的连接字符串
m_spInit()	指向数据源对象的智能指针

使用上面的函数可以完成对数据库的操作。在 14.3 节中将会以实际的例子讲解如何使用 CDataSource 类。

14.2.2　数据库访问会话类 Csession

Csession 对象表示单个数据库访问会话。一个 CDataSource 对象可以包含一个或多个会话，要创建 CDataSource 的新 Csession 对象，需要调用 Csession::Open()函数。Csession 提供 StartTransaction()、Commit()和 Abort()事务函数，分别用于启动事务、提交事务和回滚事务。表 14-2 显示了 Csession 类的成员函数。

表 14-2　Csession类的成员函数

函　数　名	功　　能
Open()	打开新会话
Close()	关闭会话
StartTransaction()	启动事务
Abort()	回滚事务
Commit()	提交事务
GetTransactionInfo()	获取事务信息
m_spOpenRowset()	指向会话的IopenRowset接口

14.2.3　记录集类 CrowSet

在 OLE DB 中，记录集是程序用于设置或获取数据的对象。CrowSet 类封装了 OLE DB 记录集对象和几个相关的接口，并提供了记录集数据的操作方法。表 14-3 显示了 CrowSet 类的成员函数。

表 14-3　CrowSet类的成员函数

函　数　名	功　　能
Close()	释放记录集和当前Irowset接口
AddRefRows()	增加与当前行相连的引用计数

续表

函　数　名	功　　能
ReleaseRows()	释放当前行句柄
Compare()	比较记录书签值
IsSameRow()	比较指定行和当前行是否相同
MovePrev()	向前移动一条记录
MoveNext()	向后移动一条记录
MoveFirst()	移动到记录集的第一条记录
MoveLast()	移动到记录集的最后一条记录
MoveToBookmark()	定位到书签处的记录，或者移动到从书签处开始指定偏移量的记录
GetDataHere()	从指定缓冲区获取数据
Insert()	插入新记录
Delete()	从记录集中删除当前记录
GetData()	获取记录集中当前行的数据
GetOriginalData()	获取最后一次从数据源获取到的数据，忽略在此期间进行过的记录编辑内容
SetData()	设置当前行的一个或多个列数据内容
GetRowStatus()	返回所有行的状态
Undo()	撤销所有从最近一次从数据源获得数据后进行的修改或撤销最后一次调用 Update 之后进行的数据修改
Update()	提交所有从最近一次从数据源获得数据后进行的修改或撤销最后一次调用 Update 之后进行的数据修改
GetApproximatePosition()	返回对应书签的行的相应的位置值
m_pRowset()	指向OLE DB的记录集对象IRowset的指针

使用上面的函数，可以实现在记录集之间定位记录、增加记录、修改记录和删除记录等操作。记录集对象可以与会话对象结合起来使用，从而实现事务处理。

14.2.4　数据表 CTable

CTable 类是个类模板，用于处理 OLE DB 中的数据表，需要与记录集结合使用才可以实现数据库表的功能。其原型为：

```
template <class TAccessor = CNoAccessor, class TRowset = CRowset >
class Table : public CAccessorRowset <T, TRowset>//CTable 类模板定义
```

其中，TAccessor 是一个访问器类，TRowset 类是一个记录集类，使用此参数用于直接访问简单的不带参数的记录集。此类只有一个函数，Open()函数用于打开指定的表，其函数原型为：

```
HRESULT Open(
  const CSession& session,          //表示打开表所使用的会话对象
  LPCTSTR szTableName,              //表示要打开的表名称
  DBPROPSET* pPropSet = NULL );     //包含属性值对的 DBPROPSET 结构的数组
HRESULT Open(
  const CSession& session,          //表示打开表所使用的会话对象
```

```
    DBID& dbid,                          //打开表的 DBID
    DBPROPSET* pPropSet = NULL );        //包含属性值对的 DBPROPSET 结构的数组
```

Open()函数的返回值为标准的 HRESULT 类型，可以通过 FAILED 或 SUCCESSED 来判断返回值是否成功。使用 CTable 模板类，可以打开不同的表，这对于检索数据库中各个表的数据非常有用。在 14.3 节中将会介绍此模板类的使用方法。

14.3　Visual C++的 OLE DB 应用实例

前面两节介绍了 OLE DB 的概念和其对应的类，本节以一个通用实例介绍如何使用 VC 开发 OLE DB 应用程序。本节实例中会显示如何使用 OLE DB 列举数据库中包含的表以及每个表的表结构。

14.3.1　创建应用程序

创建 OLE DB 程序可以使用像创建 ODBC 程序一样的步骤创建，但是本节创建的是通过表结构和表数据显示程序，因此，本程序是基于 CFormView 类的单文档工程，工程名为 OLEDBSample。要使用 OLE DB 编写数据库应用程序，在 COLEDBSampleDoc 类中增加以下变量：

```
01  //OLE DB 数据库程序的文档类定义
02  class COLEDBSampleDoc : public CDocument
03  {
04  public:
05      CString m_strConnect;                   //连接字符串
06      CDataSource m source;                   //数据源对象变量
07      CSession    m_session;                  //会话对象变量
08      CTables* m_pTableset;           //存放指定数据库中包含的数据表对象的指针
09      CColumns* m_pColumnset;         //存放指定表中存放的数据列对象的指针
10      int m_nColumns;                         //列数目
11      int m nPrevColumns;                     //原来的列数目
12      CCommand<CManualAccessor> m_Rowset; //表示命令记录集
13      struct MYBIND* pBind;                   //字段绑定，用于获取字段取值
14      CString  m_strTableName;                //表名
15      ...
16  };
```

下面代码显示了当单击工具栏上的打开按钮时，程序执行的代码。

```
01  BOOL COLEDBSampleDoc::OnOpenDocument() //打开文档时，查询记录集
02  {
03      USES_CONVERSION;
04      if (m_pTableset)                //如果表集合有效，则关闭当前所有的记录集
05      {
06          delete m_pTableset;                 //删除表集合对象
07          m_pTableset = 0;                    //设置表集合对象为 0
08      }
09      if (m_pColumnset)                       //如果列集合有效，则删除列集合
```

```
10      {
11          delete m_pColumnset;            //删除列集合对象
12          m_pColumnset = 0;               //设置列集合对象为 0
13      }
14      //释放当前会话中的记录
15      if (m_session.m_spOpenRowset != NULL)
16          m_session.m_spOpenRowset.Release();
17      if (!m_strConnect.IsEmpty())
18          m_strConnect = "";              //清空连接字符串
19      m_source.Close();                   //关闭数据库
20      if (m_source.Open() != S_OK)        //打开数据源
21      {
22          AfxMessageBox(_T("无法连接到数据源"));//显示错误消息框
23          m_strConnect = _T("");          //清空连接字符串
24          return false;                   //函数失败，返回 false
25      }
26      else                                //如果打开数据源成功
27      {
28          USES_CONVERSION;
29          if (m_session.Open(m_source) != S_OK)//打开到数据源的会话
30          {
31              AfxMessageBox(_T("无法在相应的提供程序上创建会话"));
32              //显示错误消息框
33              return false;                   //函数失败，返回 false
34          }
35          CComVariant var;          //定义组件变量类型存放连接字符串
36          //将当前打开的数据源连接字符串记录到 m_strConnect 变量中
37          m_source.GetProperty(DBPROPSET_DATASOURCEINFO,
38              DBPROP_DATASOURCENAME, &var);
39          m_strConnect = OLE2T(var.bstrVal);
40          //将组件变量类型转换为 CString 类型
41      }
42      if (FetchTableInfo())
43          //检索当前数据库中的所有表成功，函数返回 true
44          return true;
45      else
46          return false; //检索当前数据库中的所有表失败，函数返回 false
47  }
```

上面代码首先将数据重置，如原来有打开的表结构，则将其删除，m_pColumnset 对象中的列信息会删除，置为 NULL，并释放会话中打开的记录集。同时，如果数据源是打开的，则关闭原来的数据源连接。然后调用数据源的 Open()函数，连接到数据源，使用数据源对象和会话对象的 Open()函数，打开数据源上的会话对象，并获取当前的连接字符串，赋值给 m_strConnect 变量。最后调用 FetchTableInfo()自定义函数获取选择的数据库中的所有数据表信息。当运行程序时，单击工具栏上的打开按钮，会弹出如图 14-1 所示的对话框，读者需要选择要查看表结构和表记录的数据库。

14.3.2　显示数据库表

14.3.1 小节讲过单击工具栏上的打开按钮，弹出“数据链接属性”对话框，选择需要

查看数据的数据源后，就调用 FetchTableInfo()函数查询数据库表。其代码如下：

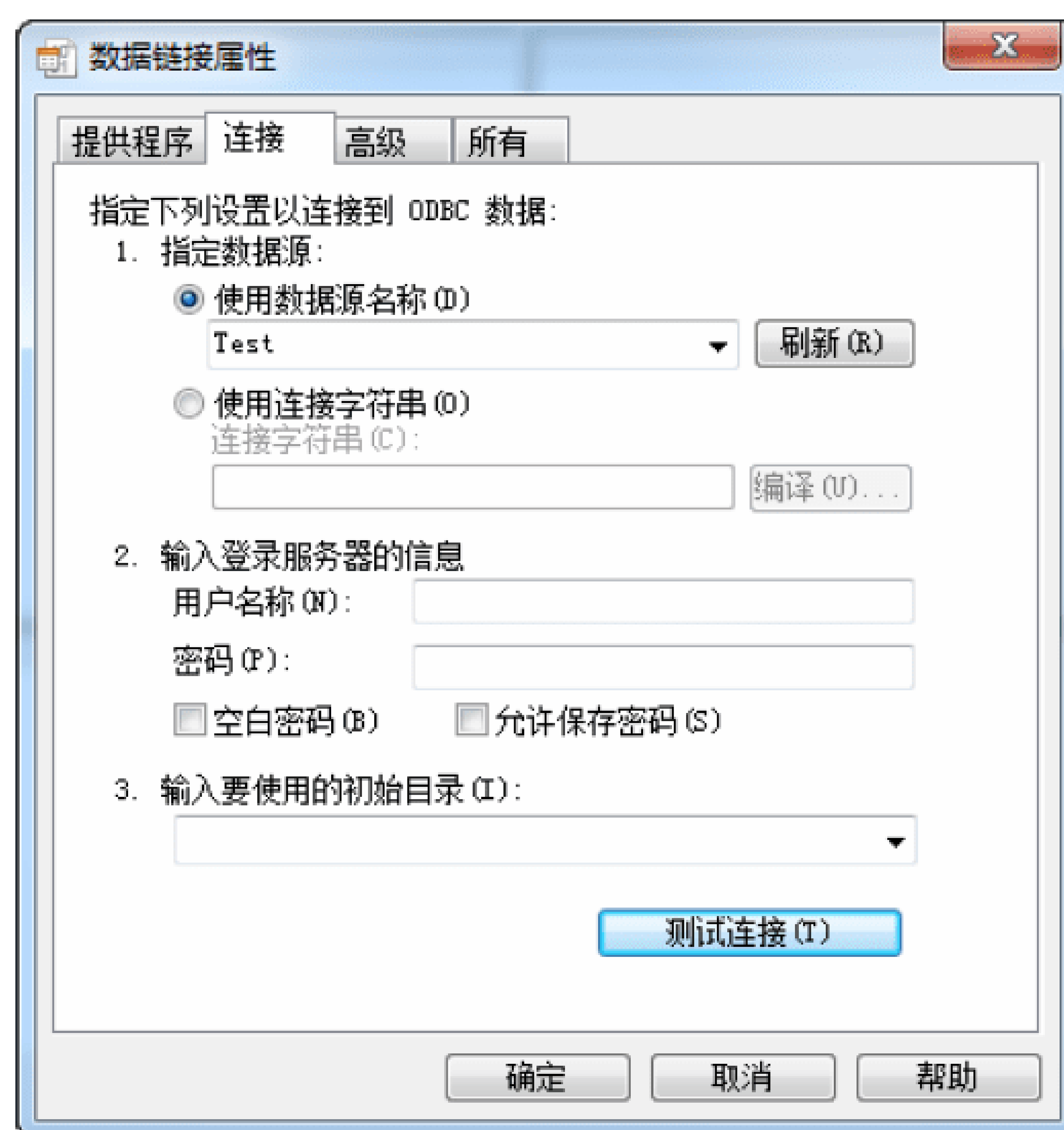


图 14-1　“数据链接属性”对话框

```
01  BOOL COLEDBSampleDoc::FetchTableInfo()//获取表信息
02  {
03      if (m_pTableset != NULL)         //如果当前表集合不为空，则清空表集合
04      {
05          delete m pTableset;          //删除表集合对象
06          m_pTableset = NULL;          //设置表集合对象为 NULL
07      }
08      m_pTableset = new CTables;       //初始化表集合
09      char lpszType[MAX_PATH];         //定义对象类型字符数组
10      strcpy(lpszType, "TABLE");       //初始化对象类型为表 TABLE
11      //strcat(lpszType, ",VIEW");     //初始化对象类型为视图，此处不用
12      //strcat(lpszType, ",SYSTEM TABLE");
13      //初始化对象类型为系统表，此处不用
14      if (m_pTableset->Open(m_session, NULL, NULL,
15          NULL, lpszType) != S_OK)
16      //打开表集合
17      {
18          //如果失败
19          delete m_pTableset;          //删除表集合对象
20          m pTableset = NULL;          //设置表集合对象为 NULL
21          return false;                //打开表集合失败，函数返回 false
22      }
23      return true;                     //函数成功，返回 true
24  }
```

上面代码显示了如何使用 CTable 类的 Open()函数打开指定会话上指定类型的对象。需要注意的是，如果需要查看视图信息，则需要在最后一个参数后增加“VIEW”字符串。获取表信息后，要在窗体中显示，则需要在视图类 COLEDBSampleView 中增加 ShowTable()函数，代码如下：

```
01  void COLEDBSampleView::ShowTable()     //显示数据库中所有的表
02  {
```

```
03      COLEDBSampleDoc*  pDoc = GetDocument();//获取应用程序对应的文档对象
04      ASSERT_VALID(pDoc);                    //诊断文档对象是否有效
05      pDoc->SetTitle(pDoc->GetDSN());        //设置当前文档的标题为数据源连接串
06      CString strDataSource = pDoc->GetDSN();//获取数据源连接串
07      strDataSource += _T(" [表]");          //在数据源连接串后增加“[表]”
08      pDoc->SetTitle(strDataSource);         //设置当前文档的标题
09      m_TablesCtrl.DeleteAllItems();         //删除数据表列表控件中的所有项
10      m_TablesCtrl.InsertColumn(0,_T("表名"),LVCFMT_LEFT,100,-1);
11      //增加显示的表名列
12      m_TablesCtrl.InsertColumn(1,_T("类型"),LVCFMT_LEFT,100,1);
13      //增加显示的类型列
14      m_TablesCtrl.InsertColumn(2,_T("目录"),LVCFMT_LEFT,100,2);
15      //增加显示的目录列
16      m_TablesCtrl.InsertColumn(3,_T("结构"),LVCFMT_LEFT,100,3);
17      //增加显示的结构列
18      m_TablesCtrl.InsertColumn(4,_T("描述"),LVCFMT_LEFT,100,4);
19      //增加显示的描述列
20      int item = 0;                          //初始化表数目为 0
21      while (pDoc->m pTableset->MoveNext() == S OK)
22          //移动到表记录集中的下一条
23      {
24          //向数据表列表框控件中新增加一条记录
25          m_TablesCtrl.InsertItem(item, pDoc->m_pTableset->m_szName);
26          //设置增加的新记录的类型字段为当前表记录的类型，在此处，肯定为 TABLE
27          m TablesCtrl.SetItem(item,1,LVIF TEXT,
28              pDoc->m pTableset->m szType,0,0,0,0);
29          //设置增加的新记录的目录字段为当前表记录的目录，即所属的数据库
30          m_TablesCtrl.SetItem(item,2,LVIF_TEXT,
31              pDoc->m_pTableset->m_szCatalog,0,0,0,0);
32          //设置增加的新记录的结构字段为当前表记录的结构
33          m TablesCtrl.SetItem(item,3,LVIF TEXT,
34              pDoc->m pTableset->m szSchema,0,0,0,0);
35          //设置增加的新记录的描述字段为当前表记录的描述
36          m_TablesCtrl.SetItem(item,4,LVIF_TEXT,
37              pDoc->m_pTableset->m_szDescription,0,0,0,0);
38          item++;                            //增加表数目索引
39      }
40  }
```

上面代码显示了如何在窗体的列表控件中显示当前选定数据库中的表。首先获取当前连接的数据源的名称，并显示在程序的标题栏中。然后，将当前数据表控件中的所有项删除，并为数据表控件增加固定的 5 列，用于描述数据表名、表类型、目录、结构和表描述。最后将 m_pTableset 集合中的所有集合遍历一遍，并将其信息显示在数据表控件中。

14.3.3　显示表定义

在 14.3.2 小节的基础上，显示完数据库表后，当用户双击其中的某个表，则在右边的列表控件中显示表的定义。以下代码显示了当用户双击某个表项时执行的操作。

```
01  void COLEDBSampleView::OnDblclkListTables(
02      NMHDR* pNMHDR, LRESULT* pResult)
03  {
04      //双击表时，同时显示表结构和表数据
```

```
05      COLEDBSampleDoc*  pDoc = GetDocument();//获取应用程序对应的文档对象
06      ASSERT_VALID(pDoc);                    //诊断文档对象是否有效
07      int nCount = m_TablesCtrl.GetItemCount();
08      //获取显示表的列表控件中的记录个数
09      for (int i = 0; i < nCount; i++)       //使用 for 循环判断当前选择的记录
10      {
11          //如果状态为选中则退出循环
12          if (m_TablesCtrl.GetItemState(i,LVIS_SELECTED))
13              break;
14      }
15      if (i < nCount)             //如果循环计数小于表个数，表示有选中的表
16      {
17          //设置文档对象的表名为选中的表
18          pDoc->m_strTableName = m_TablesCtrl.GetItemText(i,0);
19          LPCSTR lpszName;                   //定义表名变量
20          lpszName = pDoc->m_strTableName;//获取当前选择的表名
21          pDoc->FetchColumnInfo(lpszName);//检索表对应的列信息
22          ShowColumes();                     //显示表对应的列信息
23          pDoc->FetchTableData(lpszName); //检索表中的数据
24          ShowData();                        //显示表中的数据
25      }
26      *pResult = 0;                          //赋值操作结果为 0
27  }
```

在上面代码中，在 for 循环中使用 m_TablesCtrl 对象的 GetItemState()函数判断是否有被选中的表。如果有被选中的表，则获取选中的表名，获取表结构信息，显示在列表控件中，同时获取选中表中包含的数据信息，显示在记录列表控件中。以下代码是获取表的列信息的函数的实现。

```
01  //获取表的列信息
02  void COLEDBSampleDoc::FetchColumnInfo(LPCSTR lpszName)
03  {
04      if (m_pColumnset)           //如果当前列集合在使用，则清除当前的列信息
05      {
06          delete m_pColumnset;                   //删除列集合
07          m_pColumnset = NULL;                   //设置列集合为 NULL
08      }
09      m_pColumnset = new CColumns;               //实例化列集合
10      HRESULT hr = m_pColumnset->Open(m_session,
11          NULL, NULL, lpszName);
12      //打开列集合
13      if (FAILED(hr))                            //如果打开列集合失败
14      {
15          AfxMessageBox(_T("打开列信息记录集失败"));//显示错误提示消息框
16          delete m_pColumnset;                   //删除列集合
17          m_pColumnset = NULL;                   //设置列集合为 NULL
18      }
19  }
```

上面代码使用 CColumns 对象的 Open()函数打开指定会话上的指定表的列信息集合。下面是如何在列表控件中显示列信息的代码。

```
01  void COLEDBSampleView::ShowColumes()        //显示列信息
02  {
03      COLEDBSampleDoc*  pDoc = GetDocument(); //获取应用程序对应的文档对象
```

```
04       ASSERT VALID(pDoc);                          //诊断文档对象是否有效
05       m_ColumesCtrl.DeleteAllItems();         //删除列信息列表控件中的所有项
06       int column = 0;                              //定义列个数变量
07       CString strDataSource = pDoc->GetDSN();//获取数据源连接串到标题变量
08       strDataSource += _T(" - ");                  //在标题变量中增加“-”
09       strDataSource += pDoc->m_strTableName;  //在标题变量中增加表名
10       strDataSource += _T(" [列信息]");            //在标题变量中增加" [列信息]"
11       pDoc->SetTitle(strDataSource);               //设置文档标题
12       //分别增加各个列
13       m_ColumesCtrl.InsertColumn(column++,_T("列名"),
14           LVCFMT_LEFT,100,-1);
15       //增加列名列
16       m_ColumesCtrl.InsertColumn(column,_T("类型"),
17           LVCFMT_LEFT,100,column++);
18       //增加类型列
19       m ColumesCtrl.InsertColumn(column, T("长度"),
20           LVCFMT_LEFT,80,column++);
21       //增加长度列
22       m_ColumesCtrl.InsertColumn(column,_T("精度"),
23           LVCFMT LEFT,80,column++);
24       //增加精度列
25       m_ColumesCtrl.InsertColumn(column,_T("大小"),
26           LVCFMT LEFT,50,column++);         //增加大小列
27       //增加是否可为空列
28       m_ColumesCtrl.InsertColumn(column,_T("是否可为空"),LVCFMT_LEFT,50,
29           column++);
30       int item = 0;                           //初始化列数目索引为 0
31       if (pDoc->m_pColumnset == NULL)
32           return;
33       //如果文档对象的列集合为 NULL，则返回
34       while (pDoc->m pColumnset->MoveNext() == S OK)
35           //移动列集合到下一个列记录中
36       {
37           CString strValue;                   //取值变量
38           //增加新列类型，在列表控件中放在第二列
39           m ColumesCtrl.InsertItem(item,
40               pDoc->m pColumnset->m szColumnName);
41           column = 1;
42           CString strType;                    //取值类型
43           strType.Format("%d", pDoc->m_pColumnset->m_nDataType);
44           //格式化列类型字符串
45           m ColumesCtrl.SetItem(item,column++,LVIF TEXT,
46               strType,0,0,0,0);
47           //设置列类型值
48           strValue.Format(_T("%ld"),pDoc->m_pColumnset->m_nMaxLength);
49           //格式化长度字符串
50           m ColumesCtrl.SetItem(item,column++,LVIF TEXT,
51               strValue,0,0,0,0);
52           //设置列长度值
53           //格式化大小字符串
54           strValue.Format( T("%d"),
55               pDoc->m pColumnset->m nNumericPrecsion);
56           m ColumesCtrl.SetItem(item,column++,LVIF TEXT,
57               strValue,0,0,0,0);
58           //设置列大小值
59           //格式化精度字符串
```

```
60          int nOrdinal = pDoc->m pColumnset->m nOrdinalPosition;
61          strValue.Format( T("%d"),
62              pDoc->m pColumnset->m nNumericScale);
63          m ColumesCtrl.SetItem(item,column++,LVIF TEXT,
64              strValue,0,0,0,0);
65          //设置列精度值
66          if (pDoc->m_pColumnset->m_bIsNullable == false)
67              //判断列是否为 NULL
68              m ColumesCtrl.SetItem(item,column++,LVIF TEXT, T("No"),
69              0,0, 0,0);                                  //设置不可空
70          else
71              m_ColumesCtrl.SetItem(item,column++,LVIF_TEXT,_T("Yes"),0,
72              0,0,0);                                     //设置可空
73          item++;                                         //列计数索引增加 1
74      }
75      pDoc->m_nPrevColumns = pDoc->m_nColumns;            //记录原来表的列的数目
76      pDoc->m_nColumns = item;                            //记录当前表的列的数目
77  }
```

上面代码显示了如何在列表控件中显示表的结构信息。首先将当前选择的表添加在数据库名称后面，显示在程序的标题上。然后向列表控件中依次增加列名、列类型、长度、大小、精度和是否为 NULL 等 6 列。最后，遍历列对象 m_pColumnset，依次将表的每个字段的这些属性作为一条记录显示在列表控件中。程序的运行效果如图 14-2 所示。左上角的视图显示数据库中的表信息，双击表名后，右上角的视图会显示表中的字段信息，最下面的视图会将字段信息作为表头插入。

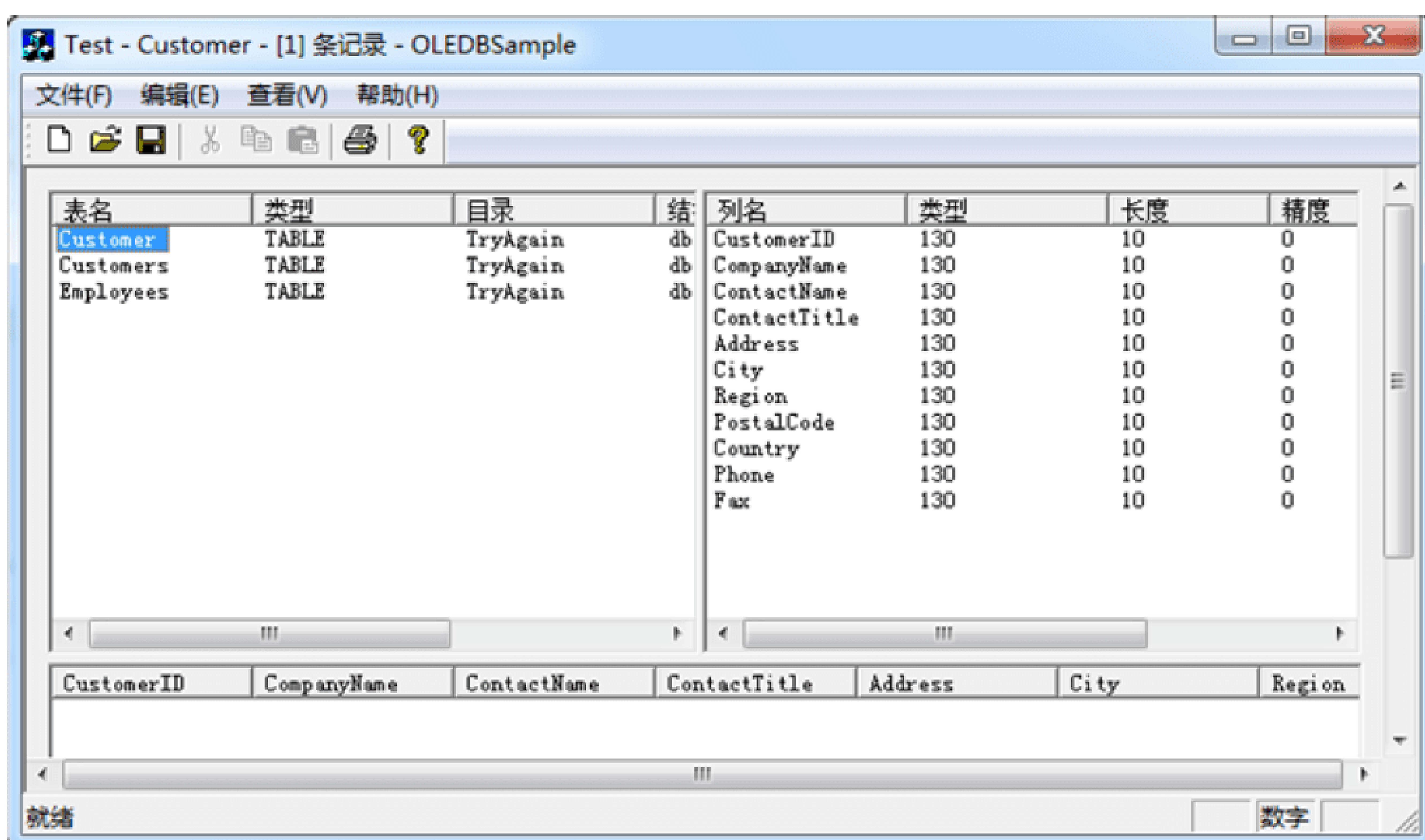


图 14-2　程序运行效果

14.4　本 章 小 结

本章介绍了在 VC 中使用 OLE DB 访问数据库的方法。本章重点和难点是掌握 OLE DB

的结构和对应的数据源 CDataSource 类、CSession 类、CRowSet 类和 CTable 类的使用。第 15 章将介绍在 VC 中访问 MySQL 数据库的技术。

14.5　习　　题

1．简述 OLE DB 与 ODBC 之间的关系。

【思路】参考 14.1.2 小节所讲的内容，然后用自己的话来描述。

2．编程使用 OLE DB 来连接第 12 章创建的数据库 TASK，然后将表 January 中的各个字段的属性显示在列表中（程序界面类似于图 14-2 所示，但窗口中只有一个 CListView，用来显示表中各个字段的属性）。

【思路】参考 14.3 节所讲的实例。

第 15 章　Visual C++中 MySQL 访问技术

MySQL 数据库是目前最流行的开放源代码并基于 SQL 的数据库。最初由瑞典 MySQL AB 公司开发并维护，现在已经被 Oracle 公司收购。MySQL 可以运行在多种平台下，其中对 Windows 开发也提供了足够的支持。本章将介绍使用 MySQL C API 连接 MySQL 数据库进行开发的方法。

15.1　MySQL C API

因为 MySQL 是使用 C 和 C++编写的，因此提供了 C API 函数，读者可以通过 MySQL C API 访问数据库，其包含在 mysqlclient 库中。本节就简要地介绍 MySQL C API 的知识及其使用。

15.1.1　MySQL C API 的数据类型

MySQL C API 中定义了一组与数据库相关的结构类型，这些类型是 MySQL 数据库中使用的对象的抽象。其与其他的数据库访问架构的类的概念是等同的。MySQL 中包含的 C API 数据类型如下。

- MYSQL 结构：表示数据库连接句柄，要执行对数据库的操作，需要使用此结构表示到数据库的连接，此句柄是不可复制的。
- MYSQL_RES 结构：表示执行查询返回的结果集。
- MYSQL_ROW 结构：代表查询结果集中的一行，使用字节字符串数组的形式组织。可以通过调用 mysql_fetch_row()函数获取。
- MYSQL_FIELD 结构：代表 MySQL 数据库中表的字段的信息，如字段名、类型和大小。通过调用 mysql_fetch_field()函数可以获取指定字段的信息。
- MYSQL_FIELD_OFFSET 结构：表示 MySQL 字段的偏移量，其值是从 0 开始，即 0 表示字段列表中的第一个字段，1 表示字段列表中的第二个字段。
- my_ulonglong 结构：表示 MySQL 中的长整型，如记录集的行数。此类型的有效范围为 0～1.84e19。

MYSQL_FIELD 结构包含了字段信息，定义如下：

```
typedef struct st mysql field {      //mysql 字段结构
    char *name;                      //列名
    char *org_name;                  //源列名
    char *table;                     //列在的表
```

```
    char *org table;               //原表
    char *db;                      //数据库
    char *catalog;                 //表结构
    char *def;                     //默认值
    unsigned long length;          //列宽
    unsigned long max_length;      //最大列宽
    unsigned int name_length;      //列名长度
    unsigned int org_name_length;  //原列名长度
    unsigned int table length;     //表名长度
    unsigned int org_table_length; //原表名长度
    unsigned int db_length;        //数据库长度
    unsigned int catalog_length;   //表结构长度
    unsigned int def_length;       //默认值长度
    unsigned int flags;            //选项，见表 15-1
    unsigned int decimals;         //浮点型精度
    unsigned int charsetnr;        //字符集
    enum enum_field_types type;    //字段类型
    void *extension;
} MYSQL_FIELD;
```

其中，flags 选项的有效取值是表 15-1 中的一项或几项的组合。

表 15-1　字段选项有效值

标　志　值	标 志 描 述
NOT_NULL_FLAG	字段不能为 NULL
PRI_KEY_FLAG	字段是主键的组成部分
UNIQUE_KEY_FLAG	字段是唯一键的组成部分
MULTIPLE_KEY_FLAG	字段是非唯一键的组成部分
UNSIGNED_FLAG	字段是无符号的
ZEROFILL_FLAG	字段使用 0 填充
BINARY_FLAG	字段是二进制形式
AUTO_INCREMENT_FLAG	字段是自增的

字段的类型 type 成员的有效取值如表 15-2 所示。

表 15-2　字段类型有效值

类　型　值	类 型 描 述
MYSQL_TYPE_TINY	短整型
MYSQL_TYPE_SHORT	短整型
MYSQL_TYPE_LONG	整型
MYSQL_TYPE_INT24	整型
MYSQL_TYPE_LONGLONG	长整型
MYSQL_TYPE_DECIMAL	浮点型 DECIMAL 或 NUMERIC
MYSQL_TYPE_NEWDECIMAL	精度数字 DECIMAL 或 NUMERIC
MYSQL_TYPE_FLOAT	FLOAT 类型
MYSQL_TYPE_DOUBLE	DOUBLE 或 REAL 类型
MYSQL_TYPE_BIT	BIT 类型，位类型
MYSQL_TYPE_TIMESTAMP	TIMESTAMP 类型，时间戳类型
MYSQL_TYPE_DATE	DATE 类型，日期类型
MYSQL_TYPE_TIME	TIME 类型，时间类型

续表

类　型　值	类 型 描 述
MYSQL_TYPE_DATETIME	DATETIME 类型，日期时间类型
MYSQL_TYPE_YEAR	YEAR 类型，年类型
MYSQL_TYPE_STRING	CHAR 类型，字符类型
MYSQL_TYPE_VAR_STRING	VARCHAR 类型，变长字符类型
MYSQL_TYPE_BLOB	BLOB 或 TEXT 类型，文本类型
MYSQL_TYPE_SET	SET 类型
MYSQL_TYPE_ENUM	ENUM 类型，枚举类型
MYSQL_TYPE_GEOMETRY	Spatial 类型，空间类型
MYSQL_TYPE_NULL	NULL-type 类型，空类型

上面列出了 MySQL 中用到的数据类型，结合这些数据类型，调用相应的 API 函数即可完成数据库访问。

15.1.2　MySQL C API 函数

MySQL 中提供了一组标准的 C　API 函数，使用这些函数可以完成对数据库的访问，包括获取数据库结构、表结构、查询数据库表中的数据以及更新数据等各种操作。表 15-3 中列出了可用的 MySQL C API 函数。

表 15-3　MySQL C API函数

函　　数	描　　述
mysql_affected_rows()	返回上次更新操作所影响的行数
mysql_autocommit()	切换 autocommit 模式，即切换是否自动提交数据库更新操作
mysql_change_user()	更改打开连接上的用户和数据库
mysql_charset_name()	返回当前连接使用的默认字符集的名称
mysql_close()	关闭当前到服务器的连接
mysql_commit()	提交事务
mysql_data_seek()	在查询结果集中查找属性行编号
mysql_debug()	执行 MySQL 调试
mysql_dump_debug_info()	将调试信息写入日志
mysql_errno()	返回上次操作 MySQL 函数返回的错误编号
mysql_error()	返回上次操作 MySQL 函数返回的错误消息
mysql_escape_string()	对字符串中的特殊字符进行转义处理
mysql_fetch_field()	获取表中下一个字段的类型
mysql_fetch_field_direct()	根据字段编号，获取表中的字段的信息
mysql_fetch_fields()	获取表中所有字段信息的数组
mysql_fetch_lengths()	获取当前行中所有列的长度
mysql_fetch_row()	定位到结果集的下一行
mysql_field_seek()	将列光标置于指定的列
mysql_field_count()	返回上次执行的 SQL 操作的列的数目
mysql_field_tell()	返回上次 mysql_fetch_field()所使用字段光标的位置
mysql_free_result()	释放结果集使用的内存
mysql_get_client_info()	以字符串形式返回客户端版本信息

续表

函　　数	描　　述
mysql_get_client_version()	以整数形式返回客户端版本信息
mysql_get_host_info()	返回描述连接的字符串
mysql_get_server_version()	以整数形式返回服务器的版本号
mysql_get_proto_info()	返回连接所使用的协议版本
mysql_get_server_info()	返回服务器的版本号
mysql_info()	返回最近执行的查询信息
mysql_init()	获取或初始化 MYSQL 结构
mysql_insert_id()	返回最近一次 AUTO_INCREMENT 列生成的 ID
mysql_kill()	杀死给定的线程
mysql_library_end()	释放 MySQL C API 库
mysql_library_init()	初始化 MySQL C API 库
mysql_list_dbs()	返回与简单正则表达式匹配的数据库名称
mysql_list_fields()	返回与简单正则表达式匹配的字段名称
mysql_list_processes()	返回当前服务器线程的列表
mysql_list_tables()	返回与简单正则表达式匹配的表名
mysql_more_results()	检查是否还存在其他结果
mysql_next_result()	在多语句执行过程中返回/初始化下一个结果
mysql_num_fields()	返回结果集中的列数
mysql_num_rows()	返回结果集中的行数
mysql_options()	为 mysql_connect()设置连接选项
mysql_ping()	检查与服务器的连接是否工作，如有必要重新连接
mysql_query()	执行指定为以 Null 结束的字符串的 SQL 查询
mysql_real_connect()	连接到 MySQL 服务器
mysql_real_escape_string()	对字符串连接中的特殊字符进行转义处理
mysql_real_query()	执行返回计数的 SQL 查询
mysql_refresh()	刷新表内容
mysql_reload()	通知服务器再次加载授权表
mysql_rollback()	回滚事务
mysql_row_seek()	使用从 mysql_row_tell()返回的值，查找结果集中的行偏移
mysql_row_tell()	返回行光标位置
mysql_select_db()	选择数据库
mysql_server_end()	释放嵌入式服务器库
mysql_server_init()	初始化嵌入式服务器库
mysql_set_server_option()	为连接设置选项
mysql_sqlstate()	返回关于最近一次错误的 SQLSTATE 错误代码
mysql_shutdown()	关闭数据库服务器
mysql_stat()	以字符串形式返回服务器状态
mysql_store_result()	将检索结果集保存到客户端
mysql_thread_id()	返回当前线程 ID
mysql_thread_safe()	如果客户端已编译为线程安全的，则返回 1
mysql_use_result()	初始化逐行的结果集检索
mysql_warning_count()	返回上一个 SQL 语句的警告数

上面的 API 函数覆盖了 MySQL 可以执行的大部分数据库访问功能，还有一些可以通

过 SQL 执行的，使用 API 函数直接执行 SQL 语句即可。如创建数据库，则执行 Create Database 即可。使用 MySQL C API 的步骤如下。

（1）调用 mysql_library_init()函数初始化 MySQL 库。

（2）调用 mysql_init()函数初始化连接处理程序，并调用 mysql_real_connect()函数指定主机名、用户名和密码等信息连接到 MySQL 服务器。

（3）调用 mysql_real_query()函数执行 SQL 语句并处理返回的结果集。对于非选择查询，如增加、修改和删除操作，可以使用 mysql_affected_rows()函数获取影响的行数。对于选择查询，则可以使用 mysql_store_result()函数或 mysql_use_result()函数和 mysql_fetch_row()函数检索结果集，使用完后需要调用 mysql_free_result()函数释放结果集。

（4）访问完数据库，调用 mysql_close()函数关闭与 MySQL 服务器的连接。

（5）调用 mysql_library_end()函数释放对 MySQL 库的使用。

（6）在使用 MySQL C API 访问数据库的过程中，可以使用 mysql_errno()函数和 mysql_error()函数获取错误代码和错误信息，并根据错误信息执行相应的处理。

根据上面的步骤，可以使用 MySQL C API 函数完成对 MySQL 数据库的访问。

15.1.3　应用程序实例

前面两小节概要介绍了 MySQL C API 中的数据类型和函数，本小节以一个实例，演示如何使用这些数据类型和 API 函数访问 MySQL 数据库。代码如下：

```
01  #include "stdafx.h"
02  #include <mysql.h>
03  //服务器参数
04  static char *server_args[] = { "mysql_test",  "--datadir=.",
05     "--key_buffer_size=32M"};
06  //数据源参数
07  static char *server_groups[] = { "mysql_test", "127.0.0.1",
08     "mysql_test", (char *)NULL};
09  int main(int argc, char* argv[])//主函数，参数是主机、用户名、密码和数据库
10  {
11     try
12     {
13        //初始化 MySQL 库
14        if(mysql_library_init(sizeof(server_args) / sizeof(char *),
15           server_args, server_groups))
16        {                        //如果初始化 MySQL 库失败
17           printf("初始化 MySQL 库失败");//输出错误提示信息
18           return 0;                 //函数返回
19        }
20        MYSQL mysql;                 //定义 MYSQL 变量
21        mysql_init(&mysql);          //初始化连接
22        if (argc == 5)        //如果输入的参数个数正确，则连接 MySQL 数据库
23        {
24           if (!mysql_real_connect(&mysql, argv[1], argv[2], argv[3],
25              argv[4], 0, NULL, 0))
26           {                     //如果连接 MySQL 数据库失败
27              printf("连接数据库失败。错误原因: %s\n",
28                     mysql_error(&mysql));
```

```
29                  return 0;               //函数返回
30              }
31          }
32          else            //如果输入的参数个数不正确，则使用默认参数连接数据库
33          {
34              if (!mysql real connect(&mysql, "127.0.0.1", "root",
35                   "123456","world", 0, NULL, 0))
36              {           //如果连接 MySQL 数据库失败
37                  printf("连接数据库失败。错误原因: %s\n",
38                          mysql error(&mysql));
39                  return 0;                               //函数返回
40              }
41          }
42          char sql[500]={0};                          //定义 SQL 查询语句数组
43          sprintf(sql, "select * from city");        //格式化 SQL 查询语句
44          if (mysql real query(&mysql, sql, strlen(sql)))
45              //执行查询操作
46          {                                           //如果执行查询失败
47              printf("执行查询操作失败。错误原因: %s\n",
48                      mysql error(&mysql));
49              return 0;                               //函数返回
50          }
51          printf("-----------------------------\n");//输出结果提示行
52          printf("查询到 artist 表中的记录: \n");      //输出结果提示信息
53          MYSQL RES* result = mysql use result(&mysql);
54          //获取查询结果集
55          MYSQL_ROW row;                              //定义记录行
56          unsigned int nFields;                       //定义字段个数
57          unsigned int i;                             //定义循环变量
58          nFields = mysql_num_fields(result);         //获取记录字段个数
59          while ((row = mysql fetch row(result))) //检索记录行
60          {
61              unsigned long *lengths;                 //定义记录长度变量
62              lengths = mysql fetch lengths(result);//获取记录长度
63              for(i = 0; i < nFields; i++)            //循环获取每个字段的值
64              {
65                  printf("[%.*s] ", (int) lengths[i],
66                      row[i] ? row[i] : "NULL");
67                  //输出记录数据
68              }
69              printf("\n");                           //每显示完一行记录，换行
70          }
71          printf("-----------------------------n");//输出结果结束行
72          mysql_close(&mysql);                        //关闭 MySQL 连接
73          mysql_library_end();                        //释放 MySQL 库
74      }
75      catch(...)                                      //如果发生异常
76      {
77          printf("执行 MySQL C API 发生异常\n");      //输出异常提示
78      }
79      return 0;                                       //执行完成，函数返回
80  }
```

上面代码演示了使用 MySQL C API 函数的过程，其中调用函数的过程与 15.1.2 小节

中介绍的方法类似，只是需要 mysql_num_fields()函数获取表中的字段个数，并调用 mysql_fetch_row()函数获取记录集行数据，并使用 mysql_fetch_lengths()函数获取字段值长度，然后显示在界面上。程序运行效果如图 15-1 所示。

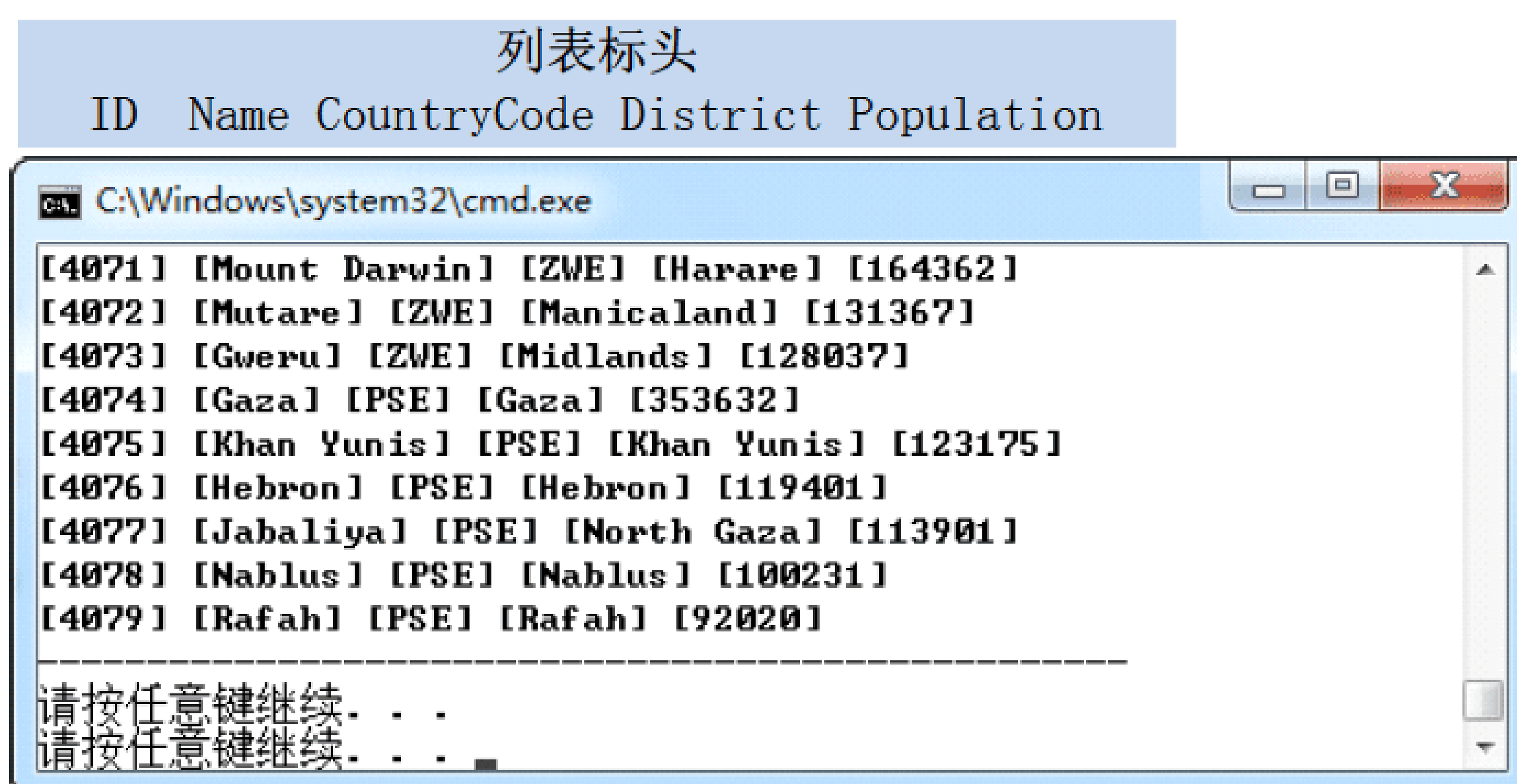


图 15-1　MySQL C API 实例运行效果

15.1.4　CDatabase 类的实现

15.1.3 小节介绍了使用 MySQL C API 函数访问 MySQL 数据库的过程，但是，通常情况下，数据库访问过程都是类似的。因此，为了简化重复代码的开发工作量，可以将相同的工作内容封装成类，每次执行类似功能时，只需调用类的主要函数即可。以下代码封装了查询表的数据库类 CDatabase。

```
01  #include "StdAfx.h"
02  #include "CDatabase.h"
03  #include <mysql.h>
04  CDatabase::CDatabase()                  //构造函数初始化 MySQL 对象
05  {
06      mysql_init(&mysql);                 //初始化 mysql 库
07  }
08  CDatabase::~CDatabase()                 //析构函数
09  {
10      Close();                            //关闭 mysql 连接
11      mysql_library_end();                //释放 mysql 库
12  }
13  CDatabase::Close()                      //关闭数据库连接
14  {
15      if(query)
16          mysql_free_result(query);
17      //如果查询有效，则释放查询结果集
18      mysql_close(&mysql);                //关闭数据库
19  }
20  //打开数据库
21  bool CDatabase::Open(char* host, char* user, char* pass, char* db)
22  {
23      //连接数据源
24      if(!mysql_real_connect(&mysql, host, user, pass, db, 0, NULL, 0))
25      {
26          //如果连接数据源失败
27          //输出错误信息
```

```
28          printf("执行查询操作失败。错误原因: %s\n", mysql_error(&mysql));
29          return false;                       //函数失败，返回
30      }
31      return true;                            //函数成功，返回
32  }
33  bool CDatabase::Execute(char* sql) //选择记录
34  {
35      if(mysql_real_query(&mysql, sql, strlen(sql)))
36          return false;
37      //执行查询失败，则返回
38      query = mysql_use_result(&mysql);   //执行查询成功，获取记录集
39      return true;                        //函数操作成功，返回 true
40  }
41  int CDatabase::GetFieldNum()            //获取字段个数
42  {
43      if (query)
44          return mysql_num_fields(query);
45      //查询有效则调用 mysql_num_fields 返回字段个数
46      return 0;                           //如果查询无效，则返回字段个数为 0
47  }
48  MYSQL_ROW CDatabase::GetRecord()        //获取记录行
49  {
50      if (query)                          //判断查询是否有效，如果有效
51      {
52          row = mysql_fetch_row(query);   //获取记录行
53          return row;                     //返回记录行
54      }
55      return NULL;                        //如果查询无效，则返回 NULL
56  }
57  void CDatabase::GetRecords()            //获取记录集
58  {
59      query = mysql_use_result(&mysql);   //使用当前查询的结果集
60  }
61  //获取记录集字段值的长度
62  unsigned long * CDatabase::GetRecordFieldLength()
63  {
64      if (query)
65          return mysql_fetch_lengths(query);
66      //如果查询有效则返回字段值长度
67      return NULL;                        //如果查询无效，则返回 NULL
68  }
69  bool CDatabase::ShowRecords(char* sql) //显示查询结果集
70  {
71      if (!Execute(sql))
72          return false;
73      //执行查询操作，如果失败，函数返回 false
74      printf("----------------------------n");   //输出结果提示行
75      printf("结果记录集: \n");                  //输出结果提示信息
76      unsigned int nFields = GetFieldNum();      //获取字段个数
77      while ((row = GetRecord()))                //循环获取记录
78      {
79          unsigned long *lengths;                //定义字段值长度变量
80          lengths = GetRecordFieldLength();      //获取记录字段值
81          for(UINT i = 0; i < nFields; i++)      //循环处理各个字段
82          {
83              //输出各个字段的值
84              printf("[%.*s] ", (int) lengths[i],
```

```
85                  row[i] ? row[i] : "NULL");
86             }
87             printf("\n");                              //显示完每条记录，换行
88         }
89         printf("-------------------------------n");  //输出信息结束提示
90         return true;                                //函数操作成功，返回 true
91     }
```

上面代码将 15.1.3 小节中的实例代码封装到了 CDatabase 类中，函数根据实例中的功能而定。在本小节 CDatabase 类的 ShowRecords()函数中，实现查询指定表中的数据并显示数据内容。读者可以根据自己的需要重新定制 CDatabase 类，关键是通过此实例掌握类封装的思想。

15.1.5　应用 CDatabase 类

15.1.4 小节介绍了如何封装 CDatabase 类，本小节介绍如何使用此类实现查询数据表的功能。创建控制台程序 MySQLCDatabaseSample，并修改 main()函数，代码如下：

```
01  #include "stdafx.h"
02  #include "CDatabase.h"
03  int main(int argc, char* argv[])        //主函数，参数是数据库连接参数
04  {
05      if (argc == 5)                      //如果输入的参数个数是 5
06      {
07          CDatabase db;                   //创建 CDatabase 类型的对象
08          //打开数据库连接
09          if (db.Open(argv[1], argv[2], argv[3], argv[4]))
10              db.ShowRecords("Select * from city");
11          //显示 artist 表中所有的记录
12      }
13      else
14          printf("输入参数错误");           //输入的参数个数不对，输出错误提示
15      return 0;                           //函数完成，返回
16  }
```

从上面代码中可以看出，应用 CDatabase 类查询指定表中的数据的方法非常简单，只需要调用 Open()函数打开到 MySQL 数据库的连接，再调用 ShowRecords()函数显示记录内容即可。这样在需要重复执行查询表内容的功能时，只需要重复使用 CDatabase 类即可，减少了重复代码。程序运行效果如图 15-2 所示。

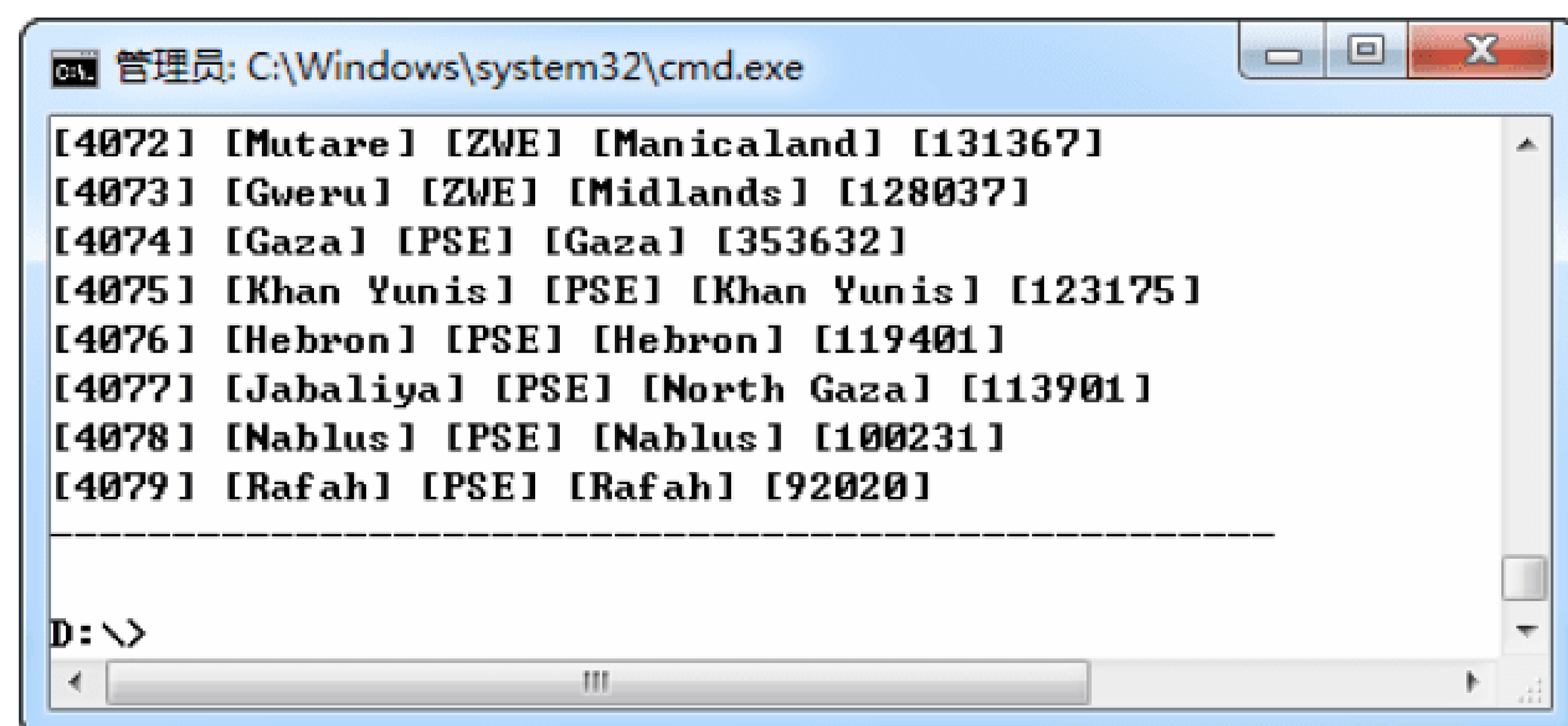


图 15-2　CDatabase 类应用运行效果

15.2　本 章 小 结

本章介绍了在 VC 中访问 MySQL 数据库的方法。重点是掌握 MySQL C API 函数访问 MySQL 数据库和使用 CDatabase 类访问 MySQL 数据库的方法。第 16 章将介绍有关通信方面的知识——Windows 套接字的编程。

15.3　习　　题

1．创建控制台应用程序，使用 MySQL 的函数，访问 MySQL 中 world 数据库的 countrylanguage 表（world 数据库是 MySQL 本身提供的数据库）。

【思路】参考 15.1.3 小节的实例。

2. 创建控制台应用程序，使用 15.1.4 小节封装的 CDatebase 类来访问 MySQL 中 world 数据库的 countrylanguage 表。

【思路】参考 15.1.5 小节的实例。

第4篇　网络编程

第 16 章　Windows 套接字编程

Windows 套接字是开放的网络编程接口，完成网络环境中的数据传输功能。本章在介绍 Windows Socket 概念的基础上，介绍套接字库函数和 WinSocket API，并讲述 MFC 对 Windows 套接字的封装。最后，以一个实例演示 Windows 套接字的编程方法。

16.1　常 见 概 念

本节介绍 Window 套接字中用到的重要概念，包括套接字及其分类和 Window Sockets 采用的编程模型——客户端/服务器模型。另外，还将介绍网络编程中的一个重要点——网络字节顺序，并且介绍了如何处理网络字节顺序与系统字节顺序之间的转换。

16.1.1　Windows Sockets 规范

Windows Sockets 规范是 Windows 平台下定义的可以兼容二进制数据传输的网络编程接口，是基于伯克利加利福尼亚大学的 BSD UNIX Sockets 的实现，当前的版本是 2.0。此规范包括 BSD 格式的 Sockets 函数和 Windows 扩展函数。使用 Windows Sockets 的应用程序可以与任何兼容 Windows Sockets API 的网络程序进行数据通信。

目前，市面上很多网络软件支持 Windows Sockets，包括传输控制协议/Internet 协议（TCP/IP）、Xerox 网络系统（XNS）、DECNet 协议、Novell 公司的 Internet 包交换和顺序包交换协议（IPX/SPX）等。虽然现在的 Windows Sockets 规范定义了提取 TCP/IP 的 Sockets，但是，任何网络协议可以通过提供自己实现的 Windows Sockets 的 DLL 版本支持 Windows Sockets。终端仿真器和电子邮件系统都是使用 Windows Sockets 典型实例。因为 Windows Sockets 是抽象于底层网络的，因此，开发人员不需要了解有关网络的知识，就可以编写运行在任何支持 Sockets 的网络上的应用程序。

因为 Sockets 编程模型使用 Internet 协议族的“通信域”，所以它是编写支持 Internet 通信应用程序首选通信方式，这也是 Socket 长足发展的原因。

16.1.2　套接字及其分类

在 Windows Sockets 规范中，使用套接字（socket）代表通信端点，在网络上通过 Windows Sockets 应用程序发送或接收数据包的对象。Socket 具有类型和名称，并具有与其类型相关的运行过程。当前，socket 通常使用 IP（Internet Protocol）与其他 socket 进行双向数据交换，所有数据流都可以同时在两个方向上进行通信。Socket 套接字分为两种类型，一种是

数据报 socket，一种是数据流 socket。

1．数据报套接字

数据报套接字，即无连接套接字，是不需要连接即可进行通信的套接字，可以向指定的 socket 发送数据报消息，也可以从指定的 socket 接收消息。提供双向的面向记录的数据流，但是不能确保数据传输的顺序，也不能确保传输的可靠性，有时会出现传输失败。通过数据报套接字传输的数据，到达目的端时，有可能打乱了发送时的字节顺序，并有可能复制传输数据，但是会控制数据的记录边界，记录小于接收端的内部大小限制。开发人员需要管理数据的顺序和可靠性，在本地局域网中可靠性比在广域网或 Internet 上高。数据报套接字的一个典型应用是，保持系统时钟与网络同步，而且使用数据报套接字可以同时向大量的网络地址广播消息。对于面向记录的数据使用数据报套接字比较合适。

2．数据流套接字

数据流套接字是基于显式连接的套接字。提供没有记录边界的双向字节数据流，具有可靠的发送顺序，没有复制数据。数据流的接收也是可靠的，适合处理大量数据的传输。客户端 Socket 请求到服务器 Sockets，服务器 Sockets 可以接收连接请求，也可以拒绝连接请求。如电话呼叫就是一个典型的数据流例子，首先呼叫方发起到被叫方的连接，被叫方可以接受连接，即接听，也可以拒绝连接，即挂断。如果接听，则链路建立了，可以进行双向数据交换，即双方可以进行通话，并且接听端会按照顺序听到说话方所说的内容，不会有重复，也不会丢失。还有一个典型的数据流 socket 的例子是 FTP（File Transfer Protocol，文件传输协议），可以用于传输任意大小的 ASCII 或二进制文件。当需要保证数据到达的可靠性和数据量比较大时，数据流套接字比数据报套接字更合适。

在一些网络层协议下，如 XNS，数据流是面向记录的记录流，而不是字节流。在更通用的 TCP/IP 协议下，数据流是面向字节的字节流。Windows Sockets 提供独立于底层协议的抽象层。

16.1.3　客户端/服务器（C/S）模型

Sockets 套接字可以用在多种架构模型下，可以用在点对点模型，如聊天程序，也可以用在远程过程调用 RPC 的通信中，但是最常用在客户端/服务器模型中。

客户端/服务器模型是常用的一种架构模型，将应用程序分成前端客户端组件和后台服务器组件。客户端组件运行在工作站上，负责从用户处接收数据，为服务器处理数据，并形成到服务器的连接。后台服务器会等待客户端的连接，当服务器接收到客户端的连接请求后，服务器会处理并返回给客户端响应信息。客户端接收到响应消息，通过用户接口呈现给用户。

目前，很多项目都设计为分布式程序，用于提高应用程序的性能。而分布式程序就是基于客户端/服务器模型的，如数据库应用程序、通信应用程序都是基于客户端/服务器模型的。在设计基于客户端/服务器模型的应用程序时，程序的性能和可扩展性是设计的关键要素。还需要考虑程序的组件和基本处理，包括数据包设计、物理部署模型、远程服务器负载以及网络带宽的分析等问题。

要提高程序性能，每个客户端应该按需处理，如果不控制客户端的连接和数据传输量，则会大大降低程序效率，因为在客户端/服务器模型下，系统瓶颈是在服务器端进行处理，因此，在设计此模型的程序时，应尽量将处理放在客户端中处理，减少服务器压力。影响客户端/服务器模型正常运行有以下因素。

- ❑ 服务器平台硬件问题：当服务器平台的硬件出现问题时，会影响客户端/服务器模型应用程序的正常运行，这个问题包括服务器硬件出现故障，也包括服务器硬件升级而需要的短暂停止，都会中断程序的正常运行。
- ❑ 服务器平台软件问题：服务器软件平台问题包括服务器应用程序本身的问题，也包括支持软件的问题，如操作系统的问题、杀毒软件的问题等，都会中断应用程序的运行，或降低服务器程序的运行效率。
- ❑ 网络问题：除了将客户端程序和服务器程序部署在同一台计算机上的情况外，当网络中断时，则应用程序就会中断运行；当网络出现阻塞时，应用程序运行效率会很低。因此，在编写客户端/服务器模型的应用程序时，需要考虑事务的处理，如当进行银行交易处理时，如果柜台客户端提交了交易请求时，服务器在返回响应前，发生网络故障，则当网络恢复时，要进行此笔交易的后期处理，也就是事务的处理。
- ❑ 应用程序问题：客户端应用程序和服务器应用程序的可靠性，对于系统的稳定性是决定性的。因此，要使系统稳定可靠的运行，需要在运行平台下进行彻底的测试。

无论哪种情况影响系统运行，应用程序服务器必须能够具有快速恢复并重新启动服务的能力。另外，在客户端/服务器模型中，客户端通过连接到服务器访问服务器的功能和数据，因此，服务器需要增加身份认证，控制客户端对服务器资源的访问，如通过身份验证、IP 地址限制等方式。这是保证系统正常运行的基本处理。总之，编写稳定、高效、安全的客户端/服务器程序可以实现资源共享和数据传输，进而完成世界互联的操作。

16.1.4　网络字节顺序

不同机器架构使用不同的字节顺序存储数据，如基于 Intel 处理器的机器与 Macintosh（Motorola）机器存储数据的字节顺序是相反的。Intel 采用的字节顺序称为“小头方式”，即低字节在前，高字节在后的方式。而标准的网络顺序是“大头方式”，即高字节在前，低字节在后的方式。两种方式的表示如下：

```
将 0x12345678 写入到以 0x0000 开始的内存中，则结果为
                大头方式        小头方式
0x0000          0x12            0x34
0x0001          0x34            0x12
0x0002          0x56            0x78
0x0003          0x78            0x56
```

其中，0x12、0x34、0x56 和 0x78 各是一个字节，组合字节顺序时，是以字 Word 为单位的，每个字包括两个字节——低字节和高字节。小头方式就是低字节在前，大头方式就是高字节在前。字之间是按照正常顺序。

一般情况下，用户不需要处理在网络上发送和接收数据的字节顺序的转换，但是在下列情况下，需要用户手动转换字节顺序。

- 用户传输的信息需要网络解释，这与发送到其他机器的数据不一样。
- 当与之通信的服务器应用程序不是 MFC 应用程序时，如果通信的两台机器使用的字节顺序不同，则需要调用字节转换。

而下列情况下，不需要用户手动调用字节转换。

- 两台机器使用相同的字节顺序，并且两端约定不进行字节交换。
- 与之通信的服务器是 MFC 应用程序。
- 用户有与之通信的服务器的源代码，因此，可以显式地说明是否转换字节顺序。
- 可以将服务器转换成 MFC 程序。

在后面会介绍到 MFC 的 CAsyncSocket 类，如果使用此类，用户必须自己管理需要的字节顺序转换。Windows Socket 提供标准化“大头方式”字节顺序模型，并提供与“小头方式”字节顺序的转换函数。而 CSocket 使用的 CArchive 类使用“小头方式”字节顺序，但是 CArchive 类处理了字节顺序转换的细节。通过在应用程序中使用标准的字节顺序，或使用 Windows Sockets 字节顺序转换函数，用户可以编写灵活的代码。

如果使用 MFC Sockets 编程，即客户端和服务器端都使用 MFC，则不需要关心字节顺序的细节。如果编写与非 MFC 应用程序进行通信的应用程序，如 FTP 服务器，则用户在将数据传入存档对象前，需要自己管理字节顺序转换。Windows Sockets 提供了 4 个转换函数，即 ntohs()、ntohl()、htons()和 htonl()，如表 16-1 所示。

表 16-1　字节转换函数

函　　数	功　　能
ntohs()	将 16 位数从网络字节顺序转换成主机字节顺序，即从“大头方式”转换成“小头方式”
ntohl()	将 32 位数从网络字节顺序转换成主机字节顺序，即从“大头方式”转换成“小头方式”
htons()	将 16 位数从主机字节顺序转换成网络字节顺序，即从“小头方式”转换成“大头方式”
htonl()	将 32 位数从主机字节顺序转换成网络字节顺序，即从“小头方式”转换成“大头方式”

下面以使用存档的 CSocket 对象的序列化函数为例，说明如何使用字节转换顺序。假定与编写的程序通信的是非 MFC 服务器应用程序，并且没有源代码。此时，非 MFC 服务器使用的是标准网络字节顺序即“大头方式”。编写的 MFC 客户端应用程序通过 CSocket 对象使用 CArchive 对象，而 CArchive 使用“小头方式”字节顺序。协议通信包为：

```
01  struct MyPack                        //自定义数据包
02  {
03      long Number;                     //流水号
04      unsigned short Command;          //命令标识
05      short ParamA;                    //参数 A
06      short ParamB;                    //参数 B
07  };
```

在 MFC 中，其定义为：

```
01  struct MyPack                                              //自定义数据包
02  {
```

```
03      long m lNumber;                        //流水号
04      short m_nCommand;                      //命令标识
05      short m_nParamA;                       //参数 A
06      short m_lParamB;                       //参数 B
07      void Serialize( CArchive& ar );        //序列化函数
08  };
```

其中，Serialize()是序列化函数，其定义为：

```
01  void MyPack::Serialize(CArchive& ar)         //自定义数据包类的序列化函数
02  {
03      if (ar.IsStoring())                      //如果是存储对象
04      {
05          ar << (DWORD)htonl(m_lMagicNumber);  //存储流水号为 DWORD
06          ar << (WORD)htons(m_nCommand);       //存储命令标识为 WORD
07          ar << (WORD)htons(m_nParamA);        //存储参数 A 为 WORD
08          ar << (WORD)htons(m_lParamB);        //存储参数 B 为 WORD
09      }
10      else                                     //如果是提取对象
11      {
12          WORD w;                              //定义 WORD 变量
13          DWORD dw;                            //定义 DWORD 变量
14          ar >> dw;                            //提取 DWORD 值
15          m_lMagicNumber = ntohl((long)dw);//将提取的 DWORD 值转换为流水号
16          ar >> w ;                            //提取 WORD 值
17          m_nCommand = ntohs((short)w);    //将提取的 WORD 值转换为命令标识
18          ar >> w;                             //提取 WORD 值
19          m_nParamA = ntohs((short)w);         //将提取的 WORD 值转换为参数 A
20          ar >> w;                             //提取 WORD 值
21          m_lParamB = ntohl((short)w);         //将提取的 WORD 值转换为参数 A
22      }
23  }
```

在本例中，非 MFC 服务器应用程序使用的字节顺序与客户端 MFC 应用程序 CArchive 字节顺序不同，因此，使用了字节顺序转换函数对数据进行了转换。当序列化函数存储数据时，首先将数据从主机顺序转换成网络顺序，然后将其存储；当序列化函数提取数据时，首先将数据从网络顺序转换成主机顺序，然后赋值给变量。

16.2　套接字库函数

Windows Socket 规范定义了一组套接字函数，用于完成 Socket 编程，还定义了一组用于处理域名、通信协议等数据的数据库函数。为了与 Windows 编程模型一致，微软提供了一组扩展的 Socket 函数。本节将分别介绍这 3 组函数。

16.2.1　套接字函数

Windows Socket 规范包含实现 Socket 编程的套接字函数，如表 16-2 所示。

表 16-2　套接字函数

套接字函数	功　　能
accept()	当有客户端 Socket 连接到服务器 Socket 上时，服务器使用此函数接收客户端的连接。此函数的返回值为新建的 Socket，维护与客户端 Socket 之间的通信，服务器可以维护多个客户端的 Socket 连接。此函数仅对面向连接的套接字有效
bind()	服务器 Socket 使用此函数绑定到本地的指定端口。当指定端口有连接到来时，可以通知服务器 Socket，再由其决定接受（accept）socket 连接还是拒绝
closesocket()	关闭 socket。对于没有设置超时时间的阻塞 socket，此函数仅仅阻塞 socket 通信
connect()	在指定 socket 上初始化连接，客户端 socket 使用此函数连接到服务器 socket
getpeername()	获取指定 socket 连接对端的名称
getsockname()	获取 socket 绑定的本地地址
getsockopt()	获取与指定 socket 相连的选项
htonl()	将 32 位的整数从主机字节顺序转换成网络字节顺序
htons()	将 16 位的整数从主机字节顺序转换成网络字节顺序
inet_addr()	将使用点号分隔的 IP 地址转换成 Internet 地址值
inet_ntoa()	将 Internet 地址值转换成使用点号分隔的 IP 地址，如 192.168.111.1
ioctlsocket()	提供 socket 控制
listen()	服务器 socket 调用此函数启动监听，开始监听是否有客户端 socket 连接
ntohl()	将 32 位的整数从网络字节顺序转换成主机字节顺序
ntohs()	将 16 位的整数从网络字节顺序转换成主机字节顺序
recv()	从无连接的或面向连接的 socket 处接收数据
recvfrom()	从无连接的 socket 处接收数据
select()	处理多个 I/O 同步
send()	发送数据到面向连接的 socket
sendto()	发送数据到面向连接或无连接的 socket
setsockopt()	设置指定 socket 的选项
shutdown()	断开 socket 的双向通信
socket()	创建通信的 socket 对象，并返回 socket 信息

上面函数列出了编写 Socket 程序时所需要的函数，要想编写高效、稳定的 socket 程序，需要深入掌握这些函数。

16.2.2　数据库函数

Windows Socket 规范中定义了一组专门用于处理域名、通信服务和通信协议等网络信息的数据库函数。使用这些函数可以获取网络能力的检测，使用 getXbyY 的函数形式，含义是通过 Y 值获取 X 值。主要包括如下 7 个函数。

（1）gethostname()函数：返回本地主机的机器名称。其函数原型为：

```
int gethostname(
    char FAR * name,              //存放返回的主机名称的缓冲区的指针
    int namelen);                 //存放缓冲区的长度
```

此函数返回本地主机的机器名称到 name 参数指定的缓冲区中，返回的主机名是以 NULL 结束的字符串。返回的主机名的形式，可能是简单的主机名也可能是完整的域名称。如果函数调用成功，则返回 0，否则返回 SOCKET_ERROR 和相应的错误代码。使用 WSAGet

LastError()函数可以获取错误代码的值。

（2）gethostbyaddr()函数：根据网络地址获取主机信息。其函数原型为：

```
struct HOSTENT FAR * gethostbyaddr (
    const char FAR * addr,              //网络字节顺序的地址指针
    int len,                            //addr 参数的长度
    int type);                          //指定地址的类型
```

以函数返回一个 HOSTENT 结构的指针，其中包含传入的网络地址对应的名称和地址，所有数据都是以 NULL 结束。如果返回值为 NULL，则表示函数调用失败。

（3）gethostbyname()函数：从主机数据库中根据主机名称获取主机信息。其函数原型为：

```
struct hostent FAR * gethostbyname (
    const char FAR * name );            //以 NULL 结束的主机名称
```

以函数返回一个指向 HOSTENT 结构的指针，结构中的 name 参数中包含查询到的结果值。如果返回值为 NULL，则表示函数调用失败。

（4）getprotobyname()函数：根据协议名称获取协议信息。其函数原型为：

```
struct PROTOENT FAR * getprotobyname (
    const char FAR * name );            //以 NULL 结束的协议名的指针
```

以函数返回 name 参数指定的包含协议名和协议号的 PROTOENT 结构指针。如果返回值为 NULL，则表示函数调用失败。

（5）getprotobynumber()函数：根据协议号获取协议信息。其函数原型为：

```
struct PROTOENT FAR * getprotobynumber (
    int number );                       //要查询的协议的主机字节顺序的协议号
```

以函数返回包含协议名和协议号的 PROTOENT 结构指针。如果返回值为 NULL，则表示函数调用失败。

（6）getservbyname()函数：根据服务器名和协议获取服务器信息。其函数原型为：

```
struct servent FAR * getservbyname (
    const char FAR * name,      //指向服务器名称的以 NULL 结束的字符串的指针
    const char FAR * proto);    //指向协议名的以 NULL 结束的字符串的指针
```

以函数返回包含服务器名称和服务号的 SERVENT 结构的指针。如果返回值为 NULL，则表示函数调用失败。

（7）getservbyport()函数：根据端口号和协议获取服务信息。其函数原型为：

```
struct servent FAR * getservbyport (
    int port,                   //指定网络字节顺序的服务的端口
    const char FAR* proto);     //协议名指针
```

以函数返回包含服务名称和服务号的 SERVENT 结构的指针。如果返回值为 NULL，则表示函数调用失败。

上面这些函数是用于获取有关网络方面的通信、协议和域名等方面的信息的数据库函数。用户可以使用这些函数查询到 socket 程序所使用的网络资源信息。在 socket 程序中需

要使用这些函数配合 socket()函数完成 socket 通信功能。

16.2.3　Windows 扩展函数

Windows Socket 规范提供了一组基于 Berkeley 套接字函数的扩展函数。这些扩展函数在实现 Socket 功能的基础上，还允许基于消息或函数进行处理，处理异步网络事件，开启重叠 I/O 功能。除了 WSAStartup()函数和 WSACleanup()函数外，编写 Socket 程序可以不使用这些扩展 API 函数，但是建议使用这些扩展函数以保持与 Windows 编程模式一致。表 16-3 中列出了有关 Socket 的 Windows 扩展函数。

表 16-3　有关Socket的Windows扩展函数

Windows 扩展函数	功　能
WSAAccept()	accept()函数的扩展版本，允许条件接收和 Socket 分组
WSAAsyncGetHostByAddr()	根据地址异步获取主机，基于消息实现
WSAAsyncGetHostByName()	根据名称异步获取主机，基于消息实现
WSAAsyncGetProtoByName()	根据名称异步获取协议信息，基于消息实现
WSAAsyncGetProtoByNumber()	根据协议号异步获取协议信息，基于消息实现
WSAAsyncGetServByName()	根据服务器名称和端口号，异步获取服务器信息，其是基于消息实现的
WSAAsyncGetServByPort()	根据端口号和协议，异步获取服务器信息，其是基于消息实现的
WSAAsyncSelect()	实现异步版本的 select()函数
WSACancelAsyncRequest()	取消异步获取系列的函数，即取消 WSAAsyncGetXByY()函数
WSACleanup()	退出底层的 Windows Socket DLL 的引用
WSACloseEvent()	销毁事件对象
WSAConnect()	Connect()函数的扩展版本，允许交换连接数据和 QOS 标准
WSACreateEvent()	创建事件对象
WSADuplicateSocket()	复制 Socket
WSAEnumNetworkEvents()	枚举网络事件
WSAEnumProtocols()	枚举当前系统中每个有效的协议信息
WSAEventSelect()	连接网络事件和事件对象
WSAGetLastError()	获取最近的 Windows Socket 错误信息
WSAGetOverlappedResult()	返回重叠操作的完成状态
WSAGetQOSByName()	根据服务名获取 QOS 参数
WSAHtonl()	Htonl()函数的扩展版本，将 32 位整数从主机字节顺序转换成网络字节顺序
WSAHtons()	Htons()函数的扩展版本，将 16 位整数从主机字节顺序转换成网络字节顺序
WSAIoctl()	ioctl 函数的重叠执行版本
WSAJoinLeaf()	增加一个结点到会话中
WSANtohl()	ntohl()函数的扩展版本，将 32 位整数从网络字节顺序转换成主机字节顺序
WSANtohs()	ntohs()函数的扩展版本，将 16 位整数从网络字节顺序转换成主机字节顺序
WSAProviderConfigChange()	接收安装服务或卸载服务的通知消息
WSARecv()	Recv()函数的扩展版本

续表

Windows 扩展函数	功　能
WSARecvFrom()	recvfrom()函数的扩展版本
WSAResetEvent()	重置事件对象
WSASend()	send()函数的扩展版本
WSASendTo()	sendto()函数的扩展版本
WSASetEvent()	设置事件对象
WSASetLastError()	设置最近的错误信息
WSASocket()	socket()函数的扩展版本。使用 WSAPROTOCOL_INFO 结构作为输入参数，并创建重叠 socket
WSAStartup()	初始化 Windows Sockets DLL
WSAWaitForMultipleEvents()	在多个事件对象上阻塞

上面这些扩展函数是对 Windows Socket 规范提供的 Socket 函数的封装，支持消息和函数处理。如在 WSAAsyncGetServByName()函数中，可以指定接收消息的对话框句柄和消息，当异步函数执行完毕后，会发送消息给对话框，读者可以在对话框中捕获相应的消息进行处理。这与 Windows 的消息编程模式是一致的。因此，Windows Socket 扩展函数的封装方便了 Socket 程序的开发。读者可以尽量使用扩展函数开发 Socket 程序。

16.3　使用 WinSock API

16.2 节介绍了套接字库函数，WinSock API 函数为开发套接字函数提供了整套处理函数。读者在调用 WinSock API 函数时，要注意函数之间的关联关系。本节将介绍使用 WinSock API 进行套接字编程的几个基本问题。

16.3.1　基本 Socket 系统调用

基本套接字系统调用主要分为套接字绑定、套接字监听、套接字连接、套接字接收、数据发送、数据接收和断开套接字这几部分的调用。

套接字绑定使用 WSPBind()函数绑定到指定的地址，其函数原型为：

```
int WSPBind (
    SOCKET s,                               //指定要绑定的套接字
    // 指定套接字要绑定的地址，指向 sockaddr 结构的指针
    const struct sockaddr FAR * name,
    int namelen,                            //指定地址参数的长度
    LPINT lpErrno );                        //返回操作执行的错误代码
```

其中，name 参数是指向 sockaddr 结构的指针，此结构定义如下：

```
sockaddr {
    _short    sa_family;                    //指定地址所属的范围
    char       sa_data[14]; };              //指定地址值
```

此函数用在无连接套接字或面向连接套接字的连接或监听前。当使用 WSPSocket()函

数创建套接字后，在命名空间中存在，但是没有为其分配地址，此函数就是用来建立套接字和本地地址的关联关系。

对于面向连接的套接字在绑定套接字后，就可以调用 WSPListen()函数启动监听，用于接收客户端套接字的连接。其函数原型为：

```
int WSPListen (
    SOCKET s,              //指定要监听的套接字
    //指定服务器可以接收的客户端套接字的个数，最大值为 SOMAXCONN
    int backlog,
    LPINT lpErrno );       //返回操作执行的错误代码
```

客户端套接字要连接到服务器套接字，则需要调用 WSPConnect()函数，此函数可以建立到对端套接字的连接，用于交换数据，并指定数据交换的服务质量。其函数原型为：

```
int WSPConnect (
    SOCKET s,                          //指定要执行连接的套接字
     //指定要连接的套接字的地址
    const struct sockaddr FAR * name,
    int namelen,                       //指定要连接的套接字地址的长度
    LPWSABUF lpCallerData,             //指定在建立连接期间要传输的用户数据的指针
    LPWSABUF lpCalleeData,             //指定在建立连接期间要接收的用户数据的指针
    LPQOS lpSQOS,                      //指向套接字流控制的指针
    LPQOS lpGQOS,                      //预留
    LPINT lpErrno );                   //返回操作执行的错误代码
```

服务器接收到客户端连接请求后，可以调用 WSPAccept()函数有条件地接收套接字连接，并返回创建的与客户端相连的套接字。其函数原型为：

```
SOCKET WSPAccept (
    SOCKET s,                          //指定要接收的客户端套接字
    struct sockaddr FAR * addr,        //指定存放接收套接字的地址的指针
    LPINT addrlen,                     //指定存放接收套接字的地址的长度
    //指定函数用于判断是否接收套接字的判断条件的执行函数
    LPCONDITIONPROC lpfnCondition,
    //指定判断函数所用的的参数
    DWORD dwCallbackData,
    LPINT lpErrno );                   //返回操作执行的错误代码
```

其中条件判断回调函数的原型为：

```
int CALLBACK ConditionFunc (
    IN  LPWSABUF          lpCallerId,
    IN  LPWSABUF          lpCaller Data,
    IN OUT   LPQOS        lpSQOS,
    IN OUT   LPQOS        lpGQOS,
    IN   LPWSABUF         lpCalleeId,
    IN   LPWSABUF         lpCalleeData,
    OUT   GROUP FAR *     g,
    IN   DWORD            dwCallbackData
);
```

其中，lpCallerId 参数和 lpCallerData 参数是包含连接地址和用户数据的参数。用户可以通过发送连接请求的地址和用户数据进行身份验证，从而确定是否接收连接请求。

连接建立完成后，就可以使用 WSPSend()函数在面向连接套接字上发送数据了。其函

数原型为：

```
int WSPSend (
    SOCKET s,                       //表示发送数据的 socket 句柄
    //指向 WSABUF 结构的数组指针，每个 WSABUF 结构包含指向缓冲区的指针和缓冲区的长度
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,            //指定 lpBuffers 数组中的 WSABUF 结构的个数
    LPDWORD lpNumberOfBytesSent,    //返回此函数发送的数据个数
    DWORD dwFlags,                  //指定发送函数的选项
    LPWSAOVERLAPPED lpOverlapped,   //指向 WSAOVERLAPPED 结构的指针
    //指向发送操作执行完毕后执行的处理函数的指针
    LPWSAOVERLAPPED COMPLETION ROUTINE lpCompletionRoutine,
    LPWSATHREADID lpThreadId,       //指向 WSATHREADID 结构的指针
    LPINT lpErrno );                //指向返回错误代码的指针
```

除了发送数据，还可以使用 WSPRecv()函数接收来自套接字的数据。其函数原型为：

```
int WSPRecv (
    SOCKET s,                       //表示接收数据的 socket 句柄
    LPWSABUF lpBuffers,             //指向 WSABUF 结构的数组的指针
    DWORD dwBufferCount,            //指定 lpBuffers 数组中的 WSABUF 结构的个数
    LPDWORD lpNumberOfBytesRecvd,   //返回此函数接收数据的个数
    LPDWORD lpFlags,                //指定并返回接收函数的选项
    LPWSAOVERLAPPED lpOverlapped,   //指向 WSAOVERLAPPED 结构的指针
    //指向接收操作执行完毕后执行的处理函数的指针
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    //指向 WSATHREADID 结构的指针，由提供程序在后续的 WPUQueueApc 调用中使用
    LPWSATHREADID lpThreadId,
    LPINT lpErrno);                 //指向返回错误代码的指针
```

使用完套接字后，还需要调用断开套接字连接，此时使用 WSPShutdown()函数可以关闭在套接字上的发送和接收操作。

```
int WSPShutdown (
    SOCKET s,               //表示要关闭的 socket 句柄
    //指定要禁止执行的操作类型，可以为 SD_RECEIVE、SD_SEND 或 SD_BOTH
    //分别表示禁止接收数据、禁止发送数据和禁止收发数据
    int how,
    LPINT lpErrno);         //指向返回错误代码的指针
```

调用完此函数后，套接字句柄还没有释放，还需要调用 WSPCloseSocket()函数释放套接字句柄。

上面这几个函数是基本套接字系统调用的函数，要开发出各种不同需求的通信程序，则需要根据情况，使用各种 Windows Socket()函数。由于篇幅原因，这里不再赘述。

16.3.2　Windows Socket 编程机理

使用 Windows Socket 编程时，需要了解几种编程方式，理解这几种编程方式的机理，从而能够根据实际情况编写适合系统需求的程序。主要包括以下几个方面：阻塞操作、非阻塞操作、异步方式和数据收发。

Windows Socket 中最简单的方式就是非阻塞操作，这也是 Windows 套接字的默认方式。在此种方式下，所有的 I/O 操作都会阻塞，直到操作完全执行完毕。因此，任何线程

在同一时间只能执行一个读写操作。如果线程正在执行接收操作，而又没有数据到达，则线程会阻塞直到有数据到达。虽然此种方式操作最简单，但是并不是最有效的方式。

与阻塞操作相反，非阻塞读写操作在执行操作后立即返回，并返回错误代码为 WSAEWOULDBLOCK，表示操作还没有完全执行完。在此种机制下，需要处理当操作执行完成后的代码，在 Windows Socket 中使用网络事件通知的方式实现。读者可以使用 WSPSelect()函数注册感兴趣的事件，则当接收到相应的网络事件，系统会为程序发送事件通知，程序可以再根据自己的需要进行数据处理。

重叠读写操作，就是同时执行多个读写操作。在 Windows Socket 中使用带有 WSA_FLAG_OVERLAPPED 选项的 WSPSocket()函数创建支持重叠读写操作的套接字。客户端使用 WSPRecv()函数或 WSPRecvFrom()函数提供接收数据的缓冲区。如果同时提供一个或多个缓冲区，则数据被放置到其中任何一个用户缓冲区中。数据发送端则使用 WSPSend()函数或 WSPSendTo()函数提供发送数据的缓冲区。重叠读写操作都会立即返回，返回 0 表示读写操作立即完成，并且使用事件对象或回调函数通知程序是否已经成功发送或接收，返回值 WSA_IO_PENDING 表示读写操作成功，但是还没有执行完毕。

Windows Socket 中使用 WSPSend()函数和 WSPSendTo()函数完成套接字数据发送功能，使用 WSPRecv()函数和 WSPRecvFrom()函数完成数据接收功能。并且这些函数可以实现自动增加数据包包头和自动减去数据包包头的功能，简化数据解析的过程。

16.3.3　面向连接的套接字编程

面向连接的套接字用于实现面向连接的协议。面向连接套接字在建立连接或接收连接请求前，需要使用 WSPBind()函数将套接字绑定到本地地址上或隐式的调用 WSPConnect()函数。即面向连接套接字是需要维护链接的，其结构类似于客户端/服务器。服务器端首先创建套件字，将其绑定到已知的本地端口，并使用 WSPListen()函数设置套接字处于监听状态，等待客户端的请求，并指定连接的队列。服务器端套接字会使用 WSPAccept()函数接收客户端套接字连接，并将其放置到客户端队列中，同时创建新的对应的套接字，与客户端套接字进行数据传输和交互。

客户端会创建相应的套接字，使用 WSPConnect()函数指定服务器地址和端口初始化连接。通常客户端不需要使用绑定操作初始化连接，而是由服务器端完成隐式的绑定。同时，如果客户端使用阻塞模式，则 WSPConnect()函数会阻塞，直到服务器端接收连接或拒绝连接才会返回。

在调用 WinSokcet API 函数时，需要注意本地地址和远程地址的用法。使用 WSPGet SockName()函数可以获取用于绑定的本地地址。尤其是在没有调用 WSPBind()函数的情况下，调用 WSPConnect()函数，最好使用此函数获取本地地址。在建立连接后，使用 WSPGetPeerName()函数可以确定远程套接字的地址。

对于面向连接的套接字，在建立连接后，就可以使用发送函数和接收函数执行相应的发送操作和接收操作。通信完成后，还需要断开连接。显式地处理套接字连接是一个良好的习惯，对于编写高效、稳定的通信程序是非常重要的一环。

断开套接字连接可以使用 WSPShutdown()函数、WSPSendDisconnect()函数或 WSP CloseSocket()函数。其中，WSPSendDisconnect()函数发送断开连接到远程套接字后断开连

接，推荐使用此函数断开套接字连接。需要注意，在 WSPRecv()函数返回 WSAECONNRESET 时，表示套接字已经断开，在判断到此返回值时，本地套接字也应该做相应的断开套接字的处理。

16.3.4　无连接套接字编程

无连接套接字，也称为数据报，是实现绑定到无连接协议的套接字，使用 WSPConnect()函数建立一个到默认目的地址的连接，从而使得套接字可以使用 WSPSend()函数和 WSPRecv()函数执行面向连接的发送和接收操作。系统会自动丢弃从一个地址而不是指定目的地址接收到的数据报。如 UDP 和 IPX 都是基于无连接的协议。

再次调用 WSPConnect()函数，即可以修改默认的目的地址，即使套接字已经“连接上”了。此时，接收到的数据报只要与新指定的目的地址不相同，系统即会丢弃数据报的内容。如果使用 WSPConnect()函数指定的地址是 NULL，则套接字会处于非连接状态，默认的远程地址是不定的，此时 WSPSend()函数和 WSPRecv()函数会执行失败，并返回 WSAENOTCONN，表示当前没有处于连接状态的套接字，但是使用 WSPSendTo()函数和 WSPRecvFrom()函数可以执行数据发送和数据接收。

WSPSendTo()函数使用异步方式发送数据到指定地址，即发送数据的套接字已经使用 WSPConnect()函数建立到指定地址的连接。其函数原型为：

```
int WSPSendTo (
    SOCKET s,                         //表示发送数据的 socket 句柄
    LPWSABUF lpBuffers,               //指向 WSABUF 结构的数组的指针
    DWORD dwBufferCount,              //指定 lpBuffers 数组中的 WSABUF 结构的个数
    LPDWORD lpNumberOfBytesSent,      //返回此函数发送的数据个数
    DWORD dwFlags,                    //指定发送函数的选项
    const struct sockaddr FAR * lpTo,
                                      //可选参数，表示指向目的套接字的地址的指针
    int iTolen,                       //指定 lpTo 参数中地址的大小
    LPWSAOVERLAPPED lpOverlapped,     //指向 WSAOVERLAPPED 结构的指针
    //指向发送操作执行完毕后执行的处理函数的指针
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    //指向 WSATHREADID 结构的指针，由提供程序在后续的 WPUQueueApc 调用中使用
    LPWSATHREADID lpThreadId,
    LPINT lpErrno );                  //指向返回错误代码的指针
```

因为此函数可以实现异步发送，因此有时需要指定发送完成时执行的处理代码，此时就需要使用 lpCompletionRoutine 参数指定回调函数。回调函数的原型为：

```
void CALLBACK CompletionRoutine (
    IN    DWORD dwError,                      //指定异步操作的完成状态
    IN    DWORD cbTransferred,                //指定发送成功的字节数
    IN    LPWSAOVERLAPPED lpOverlapped,       //异步参数
    IN    DWORD dwFlags );                    //预留
```

其中 CompletionRoutine 是自定义的函数名的占位符，该函数没有返回值。读者可以在此回调函数中执行操作。

使用 WSPRecvFrom()函数可以从无连接套接字处接收数据报，并存储源地址。其函数原型为：

```
int WSPRecvFrom (
    SOCKET s,                          //表示接收数据的 socket 句柄
    LPWSABUF lpBuffers,                //指向 WSABUF 结构的数组的指针
    DWORD dwBufferCount,               //指定 lpBuffers 数组中的 WSABUF 结构的个数
    LPDWORD lpNumberOfBytesRecvd,      //返回此函数接收数据的个数
    LPDWORD lpFlags,                   //指定并返回接收函数的选项
    struct sockaddr FAR * lpFrom,      //可选参数，表示指向源套接字的地址的指针
    LPINT lpFromlen,                   //指定 lpFrom 参数中地址的大小
    LPWSAOVERLAPPED lpOverlapped,      //指向 WSAOVERLAPPED 结构的指针
    //指向接收操作执行完毕后执行的处理函数的指针
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID lpThreadId,          //指向 WSATHREADID 结构的指针
    LPINT lpErrno );                   //指向返回错误代码的指针
```

无连接套接字编程比面向连接套接字的编程要简单些，因为不需要维护链路、处理错误重发机制等工作。使用 WSPSendTo()函数和 WSPRecvFrom()函数即可完成无连接套接字的数据的发送和接收，但是因为没有重发机制，所以对于数据传输准确率要求较高的程序，不适合使用无连接套接字编程。

16.3.5　原始套接字编程

Windows Socket 中的 TCP/IP 服务提供程序支持原始套接字编程，使用原始套接字可以编写自定义协议格式的套接字。原始套接字包括两种类型，一种是在 IP 头部分已知协议类型，第二种是实现任意协议的套接字。第一种类型的套接字有 ICMP，第二种套接字可以实现服务提供程序不支持的协议。

如果 TCP/IP 服务提供程序支持 AF_INET 族的 SOCK_RAW 套接字，则对应的协议会在 WSAEnumProtocols()协议列举函数中返回的列表中存在。如果服务提供程序允许应用程序指定创建套接字函数的 protocol 参数值为任意值，则 WSAPROTOCOL_INFO 结构的 ipProtocol 成员设置为 0。而如果使用原始套接字 SOCK_RAW，则此参数值不能指定为 0。使用原始套接字有以下几方面需要注意。

- ❑ 当应用程序发送数据报时，可以通过设置 IP_HDRINCL 选项决定是否包含 IP 头。无论 IP_HDRINCL 选项的设置是什么，应用程序总是会在接收到的数据报开始部分获取到 IP 头。当指定了以下条件，则接收到的数据报会全部复制到原始套接字中。
    - ➢ 如果套接字指定的协议号与接收到的数据报中的 IP 头中的协议号一致。
    - ➢ 如果定义套接字的本地 IP 地址与接收到的数据报中的 IP 头中指定的目的地址一致。
    - ➢ 如果套接字定义的外部地址与接收到的数据报中的 IP 头中指定的目的地址一致。
- ❑ 原始套接字会导致接收到很多不希望处理的数据报。如 PING 命令使用 SOCK_RAW 原始套接字发送 ICMP 回显请求。而当应用程序接收到 ICMP 回显响应时，其他所有 ICMP 消息，如 ICMP 的查询不到主机响应 HOST_UNREACHABLE 也会发送给应用程序。而且同一时间在同一台机器上打开多个原始套接字，则相同的数据报会发送给所有打开的套接字。应用程序必须能够识别出自己的数据报而忽略其他原始套接字的数据报，此时需要在 IP 头中使用唯一的编号类区分。

因此，从上面可以看出，原始套接字是忽略具体协议的底层协议的套接字实现。读者使用原始套接字可以实现数据截获等有关网络底层操作的功能。而要实现与应用程序或指定协议相关的数据通信，建议不要使用原始套接字。原始套接字的编程与其他套接字的编程方式是类似的。

16.4　MFC 对 WinSock API 的封装

MFC 提供两个类支持使用 Windows Sockets API 进行网络编程：类 CAsyncSocket 一对一地封装了 Windows Sockets API，类 CSocket 提供了从 CArchive 对象中序列化数据的 Sockets 功能。本节将介绍这两个类及其使用。

16.4.1　CAsyncSocket 类

为了简化套接字编程，MFC 使用 CAsyncSocket 类封装了 Windows Sockets API。使用此类可以直接使用 Sockets API 编写灵活的程序，同时又可以方便地处理网络事件。除了使用 C++将 Sockets 包装成面向对象的形式外，类还将与 Sockets 相关的 Windows 消息转换成事件通知。CAsyncSocket 类的部分函数与 Windows Socket API 的函数是一对一的，但是简化了事件通知的开发过程。如表 16-4 所示为 CAsyncSocket 类中封装的通知事件。

表 16-4　CAsyncSocket类的通知事件

事 件 名 称	触 发 情 况
OnAccept()	当服务器套接字接收到客户端套接字的连接请求，并且调用 Accept()函数接收客户端连接请求时，触发此事件
OnClose()	当连接的套接字关闭时，触发此事件
OnConnect()	当客户端套接字完成连接操作时，无论连接成功还是连接失败都会触发此事件
OnOutOfBandData()	当有带外数据需要读取时，触发此事件
OnReceive()	当套接字调用 Receive()函数从对端套接字接收数据时，触发此事件
OnSend()	当套接字调用 Send()函数向对端套接字发送数据时，触发此事件

上表中列出了 CAsyncSocket 类封装的通知事件，读者可以通过这些事件直接处理各种情况发生后的操作，而 CAsyncSocket 对象的函数与前面介绍过的 WinSocket API 有很多是对应的，读者对 WinSocket API 的原理理解透彻后，会发现使用 CAsyncSocket 类的函数进行套接字编程与使用 WinSocket API 是类似的。

16.4.2　使用 CAsyncSocket 类

因为 CAsyncSocket 类较好地封装了 Windows Socket API，因此使用 CAsyncSocket 类的步骤与直接使用 WinSocket API 的步骤类似，步骤如下。

（1）构造 CAsyncSocket 对象并使用对象创建底层的 SOCKET 句柄。代码如下：

```
01   CAsyncSocket* pSocket = new CAsyncSocket;   //创建 CAsyncSocket 对象
02   pSocket->Create(6650, SOCK_DGRAM );         //创建无连接套接字
```

上面代码使用 Create()函数指定端口和地址，创建了一个无连接套接字。

（2）如果套接字是客户端，则使用 Connect()函数连接到服务器套接字；如果套接字是服务器套接字，则调用 Listen()函数启动监听，等待接收来自服务器端的套接字。当监听到来自服务器端的套接字后，调用 Accept()函数接收套接字。接收连接后，可以通过密码或地址等信息验证客户端套接字的身份。

（3）通过 CAsyncSocket 对象的成员函数执行双向的数据通信。

（4）销毁 CAsyncSocket 对象。如果套接字对象在堆栈上，则析构函数会在套接字变量超出作用范围后，自动执行；如果使用 new 操作符创建套接字，则需要使用 delete 操作符销毁对象。析构函数调用对象的 Close()函数关闭套接字连接。

在创建 CAsyncSocket 对象时，对象封装了 Windows 的 SOCKET 句柄并提供了在此句柄上的操作。当读者使用 CAsyncSocket 对象时，必须要处理阻塞操作、接收和发送机制使用的字节顺序，以及 Unicode 字符集和多字节字符集的转换等问题。

16.4.3　CSocket 类

CSocket 类派生于 CAsyncSocket 类，通过 MFC 的 CArchive 对象提供 Sockets 的存档功能，使用过程比 CAsyncSocket 模型要简单得多。CSocket 类从 CAsyncSocket 类继承了很多封装了 Windows Sockets API 的成员函数。因此，使用 CSocket 类一般不需要深入了解 Socket 编程。更方便的是，CSocket 提供了 CArchive 的同步操作，实现了 Socket 通信的文档序列化。可以使用 MFC 序列化协议发送数据和接收数据。网络传输层会将数据分割成大小合适的数据包，CSocket 类可以处理包装和解包工作。在 CAsyncSocket 类的基础上，CSocket 类提供了以下几个函数。

- ❑ IsBlocking()函数：此函数可以确定当前是否在执行一个阻塞调用。
- ❑ FromHandle()函数：返回一个指向 CSocket 对象的指针，其中存放了 SOCKET 句柄。使用此 SOCKET 句柄，可以使用 WinSocket API 执行其他套接字函数。
- ❑ Attach()函数：可以将一个 SOCKET 句柄附加到 CSocket 对象上。
- ❑ CancelBlockingCall()函数：可以取消当前的阻塞操作。

16.4.4　使用 CSocket 类

使用 CSocket 对象，首先需要创建 CSocket 对象，并将几个 MFC 类对象关联起来。下面的程序中，服务器 Sockets 和客户端 Sockets 除了第（3）步，其余每步都必须执行，其中每种 Sockets 类型需要不同的操作。当运行时，通常服务器应用程序先启动准备好并“监听”，然后客户端应用程序发起连接。如果客户端试图连接时，服务器没有准备好，则需要客户端程序稍后再试。使用 CSocket 在服务器端 Socket 和客户端 Socket 之间进行通信的流程图如图 16-1 所示。

结合图 16-1，使用 CSocket 具体步骤如下所述。

（1）构造 CSocket 对象。

（2）使用对象创建底层的 SOCKET 句柄。对于客户端 CSocket 对象，使用默认参数调用 Create()方法就可以。对于服务器 CSocket()对象，必须在 Create()方法中指定端口，图 16-1 中服务器列的第（2）步中，Create()方法的参数 nPort 就是服务器要监听的端口号。但是 CArchive 不能与数据报套接字一起使用。如果要以数据报套接字的方式使用 CSocket，

则不能使用档案文件。因为数据报是不可靠的，不能保证数据到达、数据不重复和顺序，因此，不能通过档案文件与序列化兼容。如果使用带有 CArchive 对象的 CSocket 类操作数据报套接字时，MFC 会抛出错误。

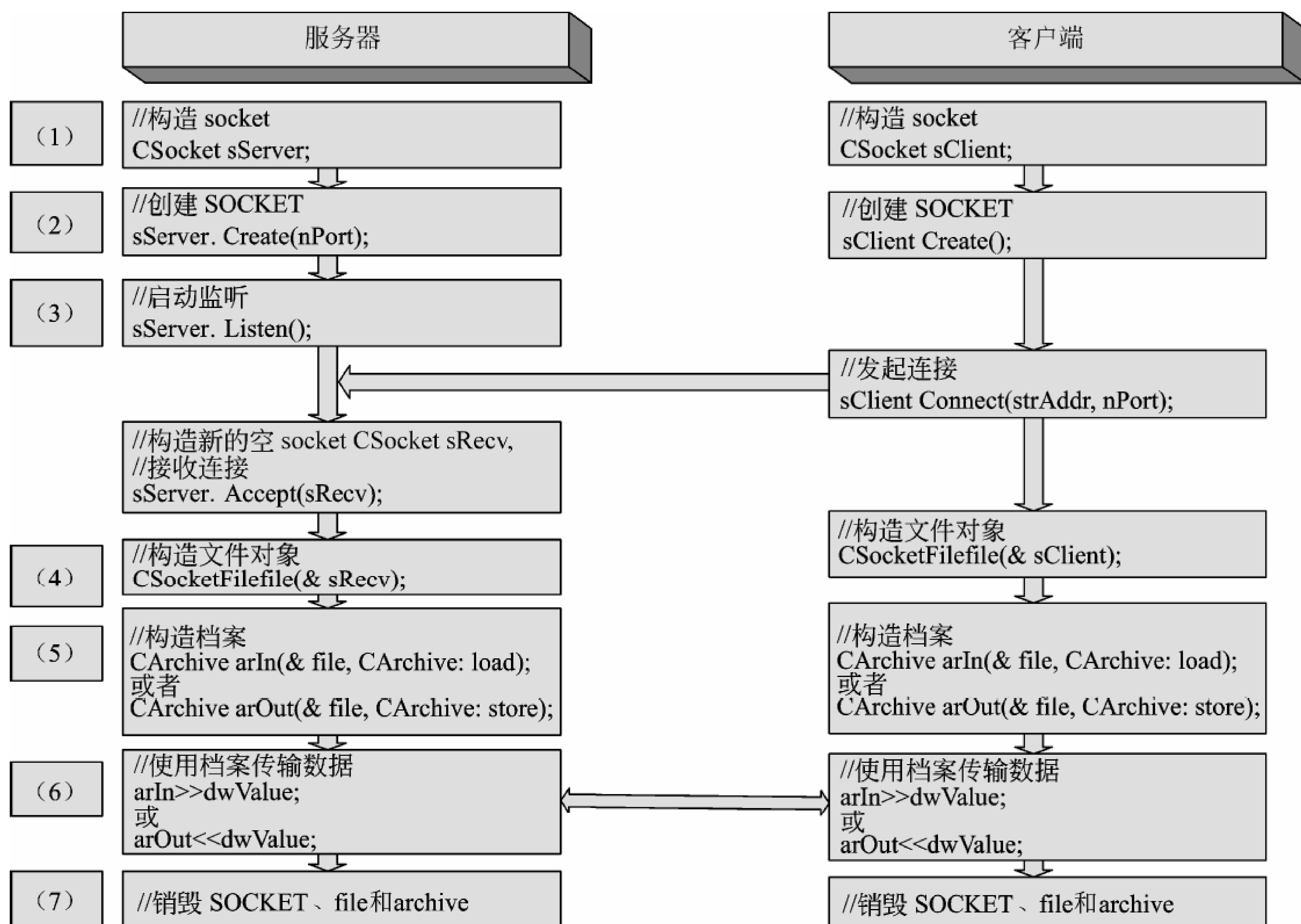


图 16-1　使用 CSocket 在服务器端 Socket 和客户端 Socket 之间的通信步骤

（3）如果 socket 是服务器，则调用 CAsyncSocket::Listen 开始“监听”客户端的连接。如果 socket 是客户端，则调用 CAsyncSocket::Connect 连接 socket 对象到服务器 socket。图 16-1 中客户端列的第（3）步中，Connect 方法的 strAddr 参数，表示要连接的服务器的地址，可以是机器地址，也可以是 Internet 协议地址，即 IP 地址。机器地址可以使用类似域名的方式，如 ftp.myServer.com，而 IP 地址使用点号分隔符的格式，如 192.168.111.1。Connect()函数会首先检查地址是否是 IP 地址的形式，如果不是，则会将地址作为机器地址处理。参数 nPort 是表示要连接的机器的端口，此处应该与服务器端 SOCKET 调用 Create()方法使用的端口一致。

当服务器端收到连接请求，如果接收，则调用 CAsyncSocket::Accept()函数。Accept 成员函数需要一个新 socket 的引用，即空 CSocket 对象。用户在调用 Accept()函数前，需要构造对象。如果 socket 对象超出范围，则连接会被关闭。MFC 会连接新对象到 SOCKET 句柄。

（4）创建 CSocketFile 对象，使其与 CSocket 对象相连。

（5）创建用于装载（接收）或存储（发送）数据的 CArchive 对象，使其与 CSocketFile 对象相连。

（6）使用 CArchive 对象在客户端和服务器 sockets 之间传输数据。要注意，CArchive

对象只能单方向存取数据。因此，有些情况下，需要使用两个 CArchive 对象，一个用于发送数据，另外一个用于接收。在接收连接和建立档案后，用户可以通过验证密码等操作，保证数据传输的安全性。

（7）销毁档案对象和 socket 对象。

CArchive 类为 CSocket 类提供 IsBufferEmpty 成员函数以确保接收所有数据。如果缓冲区中包含多条数据消息，则需要循环处理，读取全部数据消息，并且清空缓冲区。否则，有数据可以接收的下条通知会不确定地延期。

16.5　MFC Socket 实例

本节将以一个实例，说明 MFC Socket 的使用方法。此实例包含两个程序，一个是服务器端，一个是客户端。一个服务器可以接受多个客户端的连接，并可以向客户端发送数据。下面是服务器端套接字的源代码。

```
01  class CSocketServer : public CAsyncSocket
02  {
03  public:
04      CSocketServer();
05      virtual ~CSocketServer();
06  public:
07      //删除客户端连接
08      void DeleteRemoteSocket(CSocketClient* pSock);
09      //获取指定客户端在链表中的位置
10      POSITION GetRemoteSocketPos(CSocketClient* pSock);
11      //获取 SOCKET 对应的客户端变量
12      CSocketClient* GetRemoteSocket(int pSock);
13      //客户端 SOCKET 链表
14      CPtrList m_clientList;
15      //接收消息的窗口句柄对象
16      HWND m_hMsgWnd;
17  };
```

上面是服务器端 SOCKET 的头文件，其中定义了 CSocketServer 类，此类继承自 CAsyncSocket 类。下面是此类的实现代码。

```
01  CSocketServer::CSocketServer()
02  {
03  }
04  CSocketServer::~CSocketServer()
05  {
06      //如果客户端链表不为空
07      while (!m_clientList.IsEmpty())
08      {
09          //获取首个客户端对象
10          CSocketClient* client=(CSocketClient*)
11                                  m_clientList.RemoveHead();
12          client->Close();
13          delete client;
14      }
15      //移除所有客户端
16      m_clientList.RemoveAll();
17      if (m_hSocket!=INVALID_SOCKET)
```

```
18          Close();
19    }
20    //接收客户端回调函数
21    void CSocketServer::OnAccept(int nErrorCode)
22    {
23        char* pLog=new char[200];                //定义消息日志
24        if (nErrorCode)
25        {
26            if (nErrorCode==WSAENETDOWN)         //如果错误是网络故障
27                sprintf(pLog, "网络故障!");
28            else
29                sprintf(pLog, "FD ACCEPT 未知错误");
30            return;
31        }
32        else
33        {
34            sockaddr address;
35            CString IPaddr;
36            UINT port;
37            int address_len;
38            address_len=sizeof(address);
39            CSocketClient* pSocket=new CSocketClient();
40            pSocket->m_hMsgWnd=m_hMsgWnd;
41            if (this->Accept(*pSocket,&address,&address_len))
42            {
43                //设置读写关闭事件
44                pSocket->AsyncSelect(FD_WRITE|FD_READ|FD_CLOSE);
45                //获取客户端 IP 地址和端口号
46                pSocket->GetPeerName(IPaddr,port);
47                pSocket->m_strIP=IPaddr;
48                //将客户端对象增加到链表中
49                m_clientList.AddTail(pSocket);
50                sprintf(pLog, "接收客户端连接。IP=%s;端口=%d", IPaddr, port);
51            }
52            else
53            {
54                int Error=GetLastError();
55                if (Error==WSAECONNREFUSED)
56                    sprintf(pLog, "拒绝连接");
57                else
58                {
59                    wsprintf(pLog,"WSAAccept 失败，错误代码: %d",Error);
60                }
61                delete pSocket;
62                return;
63            }
64        }
65
66        if (m_hMsgWnd!=NULL)
67            ::SendMessage(m_hMsgWnd, WM_SOCKET_LOG, (WPARAM)pLog,
68                          strlen(pLog));
69
70        CAsyncSocket::OnAccept(nErrorCode);
71    }
72    void CSocketServer::OnClose(int nErrorCode)
73    {
74        while (!m_clientList.IsEmpty())
75        {
76            CSocketClient* client=(CSocketClient*)
```

```
77                                m_clientList.RemoveHead();
78          client->Close();
79          delete client;
80      }
81      m_clientList.RemoveAll();
82
83      if (m_hSocket!=INVALID_SOCKET)
84          Close();
85      CAsyncSocket::OnClose(nErrorCode);
86  }
87  void CSocketServer::DeleteRemoteSocket(CSocketClient* pSock)
88  {
89      pSock->Close();
90      POSITION pos=m_clientList.GetHeadPosition();
91      POSITION temp;
92      while(pos!=NULL)
93      {
94          temp=pos;
95          CSocketClient* client= (CSocketClient*)
96                                m_clientList.GetNext(pos);
97          if (client==pSock)
98          {
99              m_clientList.RemoveAt(temp);
100             client->Close();
101             delete client;
102             break;
103         }
104     }
105     return ;
106 }
```

上面代码是处理服务器端 SOCKET 的实现，主要用于处理 ACCEPT 事件，接收到 OnAccept 事件后，会接收客户端连接并存入链表。下面代码是客户端 SOCKET 的实现。

```
01  CSocketClient::CSocketClient()
02  {
03      m_nLength=0;                          //数据长度初始化为 0
04      memset(m_szReceBuf,0,sizeof(m_szReceBuf));
05      memset(m_szSendBuf,0,sizeof(m_szSendBuf));
06      m_bConnect=false;                     //初始化连接状态变量为 false
07      m_hWnd=NULL;                          //初始化窗口句柄为 NULL
08      m_strHost.Empty();                    //初始化主机字符串为空
09      m_strIP.Empty();                      //初始化 IP 字符串为空
10  }
11  CSocketClient::~CSocketClient()
12  {
13      if (m_hSocket!=INVALID_SOCKET)  Close();//如果 SOCKET 句柄有效，则关闭
14  }
15  void CSocketClient::OnSend(int nErrorCode)
16  {
17      //发送缓冲区中的数据
18      int nSendBytes = Send(m_szSendBuf,strlen(m_szSendBuf),0);
19      char* pLog=new char[200];
20      sprintf(pLog, "客户端发送%d 个数据", nSendBytes);
21      //如果窗体不为空，则发送日志显示消息
22      if (m_hMsgWnd!=NULL)
23          ::SendMessage(m_hMsgWnd, WM_SOCKET_LOG, (WPARAM)pLog,
24                        strlen(pLog));
```

```
25      memset(m_szSendBuf,0,sizeof(m_szSendBuf));
26      //设置读和关闭事件
27      AsyncSelect(FD_READ|FD_CLOSE);
28  }
29  void CSocketClient::OnReceive(int nErrorCode)
30  {
31      m_nLength=Receive((void*)m_szReceBuf,MAXSOCKBUF,0);
32      m_szReceBuf[m_nLength]=0;
33      char* recvBuf=new char[MAXSOCKBUF];
34      sprintf(recvBuf,(const char*)m_szReceBuf,m_nLength);
35      //如果日志窗体不为空，则发送日志显示消息
36      if (m_hMsgWnd!=NULL)
37          ::SendMessage(m_hMsgWnd, WM_SOCKET_RECEIVE, (WPARAM)recvBuf,
38                       strlen(recvBuf));
39      CAsyncSocket::OnReceive(nErrorCode);
40  }
41  void CSocketClient::OnConnect(int nErrorCode)
42  {
43      char* pLog=new char[200];
44      if (nErrorCode == 0)
45      {
46          sprintf(pLog, "连接服务器成功");
47          m_bConnect = TRUE;
48      }
49      else
50          sprintf(pLog, "连接服务器失败，错误代码=%d", nErrorCode);
51      if (m_hMsgWnd!=NULL)
52          ::SendMessage(m_hMsgWnd, WM_SOCKET_LOG, (WPARAM)pLog,
53                       strlen(pLog));
54  }
55  void CSocketClient::Init()                    //客户端 SOCKET 初始化
56  {
57      memset(m_szReceBuf,0,sizeof(m_szReceBuf));
58      m_nLength=MAXSOCKBUF;                     //设置每次接收的数据长度
59  }
```

上面代码分别处理了客户端 SOCKET 的函数，主要是处理客户端 SOCKET 连接、发送数据、接收数据和关闭连接等操作。客户端程序和服务器程序的运行效果分别如图 16-2 和图 16-3 所示。

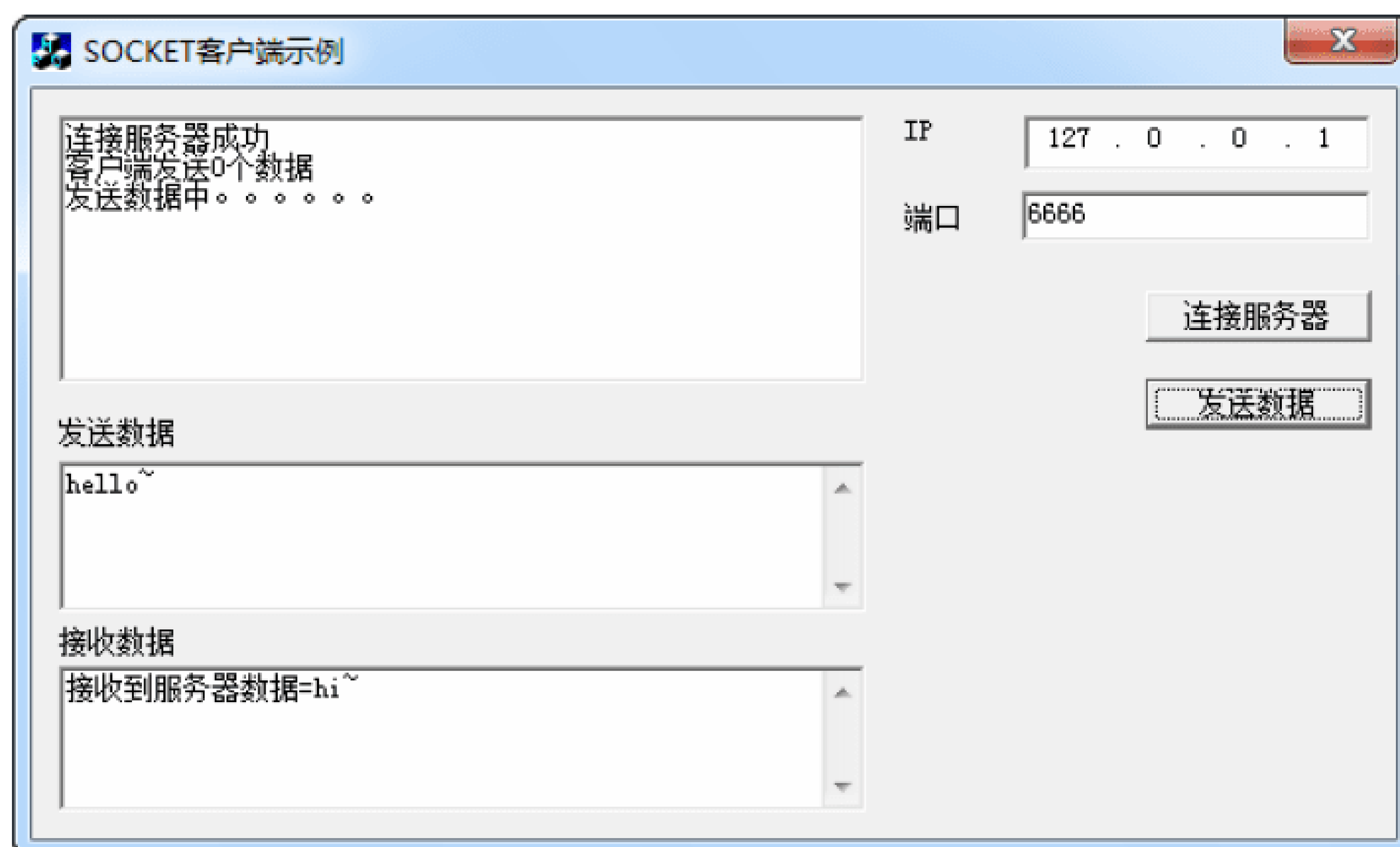


图 16-2　客户端 SOCKET 运行效果



图 16-3　服务器 SOCKET 运行效果

16.6　本 章 小 结

本章介绍了 Windows 套接字的编程知识，重点是掌握使用 MFC 的 CAsyncSocket 类和 CSocket 类编写套接字程序，难点是理解套接字的概念和机制。第 17 章将介绍邮槽和管道的编程。

16.7　习　　题

1．什么是网络字节顺序？与主机字节顺序的区别是什么？

【思路】参考 16.1.4 小节的内容。

2．在 16.5 节中使用了 MFC 封装的类 CAsyncSocket 和 CSocket 编写了一个基于 C/S 模式的聊天程序。这两个类是对 WinSock API 的封装，那么能否将类的成员函数对应到 WinSock API 上呢？如果编写网络程序的流程是固定的话，那么就可以用 WinSock API 来取代类 CAsyncSocket 和 CSocket，来实现同样的功能，尝试完成它。

【思路】理清 16.5 节中实例的具体实现流程，再找到实现同样功能的 WinSock API 来编写。需要结合 16.3 节对 WinSock API 的介绍。

第 17 章　邮槽与管道

邮槽和管道是完成进程间通信的重要方法，用于在进程之间传输各种类型的数据。而其中邮槽是单向的数据传输通道，管道又分为匿名管道和命名管道，匿名管道是本地的双向数据传输通道，命名管道是支持网络和本地两种方式的双向数据传输通道。本章的重点是讲解如何选择适当的邮槽和管道。

17.1　邮　　槽

邮槽，即类似于生活中的邮箱，进程之间可以将要发送的数据内容传递给邮槽，而相应的进程也可以从指定的邮槽中获取要得到的数据，进而对数据进行相应的处理。本节将主要介绍有关邮槽的使用方式。

17.1.1　实施细节

邮槽是进程间通信（InterProcess Communications，IPC）的一种方式，Windows 应用程序可以在邮槽中存储消息，邮槽的所有者可以从其中获取存储的消息，这些消息在网络上发送给指定的计算机或指定域上所有的计算机，其中域是共享一个组名的工作站和服务器的分组。使用邮槽，进程可以广播消息到多个进程中，即类似于 Socket 中的多播的概念。邮槽使用数据报方式在进程间发送数据，数据报是网络上发送的小的信息包。像收音机或电报一样，数据报不支持收到确认信息，并且不能保证数据报一定可以接收到。就像干扰信号可能导致收音机或电视丢失信号一样，有些情况会使数据报无法传输到特殊的目的地。

在 Windows 操作系统中邮槽由模拟文件使用，放置在内存中，可以使用标准的 Win32 文件函数访问邮槽。邮槽消息中的数据可以是任何形式的。与磁盘文件不同的是邮槽是临时存储的，并不是永久存储在文件中的。当邮槽的所有句柄都被关闭后，邮槽和包含的所有数据会被删除。

创建邮槽的进程称为邮槽服务器，负责管理邮槽。当进程创建邮槽时，会接收到邮槽句柄，进程使用邮槽句柄从邮槽中读取消息，只有创建邮槽或通过其他机制获取句柄的进程可以从邮槽中读数据。但是，进程不能创建远程的邮槽。

邮槽客户端是向邮槽中写消息的进程。任何进程，可以使用邮槽的名称向其中写消息。在邮槽中，新消息放在已存在的消息后面，即邮槽采用先进先出的存储结构。

当进程创建邮槽时，邮槽名称必须使用下面的形式：

```
\\.\mailslot\[path]name
```

邮槽名称需要的元素包括：两个反斜线开始，一个句号，一个反斜线，邮槽关键字 mailslot，以及一个结束反斜线。名称是不区分大小写的。邮槽名称前面可以加上路径，可以指定一个或多个目录的名称，使用反斜线分隔。如用户要从 zhangsan、lisi 和 wangwu 那里分别获得日志信息，用户的邮槽应用程序可以为每个数据发送者创建一个个人邮槽，代码如下：

```
\\.\mailslot\logs\zhangsan log
\\.\mailslot\logs\lisi log
\\.\mailslot\logs\wangwu_log
```

要将消息放到邮槽中，进程使用名称打开邮槽。在本地计算机上，要向邮槽中写消息，进程可以使用与创建邮槽的相同形式的邮槽名称。而在远程计算机上，更通用的邮槽名称形式如下：

```
\\计算机名\mailslot\[path]name
\\域名\mailslot\[path]name
\\*\mailslot\[path]name
```

从上面可以看出，邮槽名称的通用格式是计算机名+邮槽关键字 mailslot+邮槽名称。而英文句号代表邮槽是本地邮槽，计算机名称表示创建邮槽的计算机名，域名表示邮槽要在哪个域中广播消息，*号表示在主域中的邮槽。

17.1.2　邮槽服务器

要实现邮槽程序，需要创建一个邮槽服务器应用程序，负责创建邮槽。邮槽服务器从客户端中读取数据的过程与从文件中读取数据的过程是类似的。其过程如下。

（1）调用 CreateMailslot()函数，创建指定名称的邮槽，并返回邮槽服务器句柄，在后续的操作中，使用此句柄可以完成对邮槽的读写操作。其函数原型如下：

```
HANDLE CreateMailslot(
    LPCTSTR lpName,           //指定邮槽名称的非空字符串，格式与 17.1.1 小节中相同
    DWORD nMaxMessageSize,   //指定邮槽单个消息的最大字节数
    //读取数据的超时时间，MAILSLOT_WAIT_FOREVER 表示无限等待
    DWORD lReadTimeout,
    //指定返回的句柄是否可以被其子进程继承
    LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

如果创建邮槽成功，则返回值是服务器邮槽句柄。如果邮槽已经存在，则创建时会发生错误，返回值为 INVALID_HANDLE_VALUE。

（2）使用返回的邮槽句柄，调用 ReadFile()函数，从客户端邮槽中读取数据。在邮槽中，只有服务器邮槽可以从邮槽中读取数据。调用 ReadFile()函数的方法与前面介绍过的从文件中读取数据的方法是相同的。

（3）调用 CloseHandle()函数关闭邮槽句柄。使用完邮槽后，不要忘记关闭邮槽句柄，以释放资源。如果不调用此函数，则邮槽所属的服务器进程退出后，系统会自动关闭邮槽句柄。关闭邮槽后，邮槽中的任何消息都会被删除，同时关闭对应的邮槽客户端句柄，系统也会在内存中删除邮槽。

17.1.3　邮槽客户端

要实现邮槽客户端功能，需要编写邮槽客户端应用程序。打开到服务器邮槽的句柄，并向邮槽中写入数据。邮槽客户端的处理步骤如下。

（1）调用 CreateFile()函数打开到服务器邮槽的连接，返回对应的邮槽句柄。

（2）调用 WriteFile()函数向邮槽写入数据。

（3）调用 CloseHandle()函数关闭客户端邮槽句柄。

17.1.4　其他功能函数

除了创建打开邮槽和读写邮槽函数外，Win32 中还包括一些其他的邮槽 API，用于获取和设置邮槽信息、邮槽句柄属性等。主要包括以下几个邮槽 API。

（1）GetMailslotInfo()函数可以获取有关指定邮槽的信息。其函数原型如下：

```
BOOL GetMailslotInfo(
    HANDLE hMailslot,                    //获取信息的邮槽句柄
    LPDWORD lpMaxMessageSize,            //邮槽消息最大的字节数
    LPDWORD lpNextSize,                  //下一条邮槽消息的字节数
    LPDWORD lpMessageCount,              //邮槽中待读取的消息个数
    LPDWORD lpReadTimeout );             //邮槽读操作的超时时间设置
```

（2）SetMailslotInfo()函数设置邮槽读操作的超时时间值。其函数原型如下：

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,                    //要设置读操作超时时间值的邮槽句柄
    DWORD lReadTimeout );                //要设置的读操作的超时时间值，单位是毫秒
```

（3）GetFileTime()函数可以获取邮槽创建的日期和时间。其函数原型如下：

```
BOOL GetFileTime(
    HANDLE hFile,                        //要获取信息的邮槽句柄
    LPFILETIME lpCreationTime,           //邮槽创建的时间
    LPFILETIME lpLastAccessTime,         //邮槽最后一次访问的时间
    LPFILETIME lpLastWriteTime);         //邮槽最后一次写操作的时间
```

（4）SetFileTime()函数可以设置邮槽创建的日期和时间。其函数原型如下：

```
BOOL SetFileTime(
    HANDLE hFile,                        //要获取信息的邮槽句柄
    CONST FILETIME *lpCreationTime,      //要设置的创建时间
    CONST FILETIME *lpLastAccessTime,    //要设置的最后一次访问时间
    CONST FILETIME *lpLastWriteTime);    //要设置的最后一次写操作的时间
```

17.1.5　邮槽应用示例

前面几小节简单地介绍了与邮槽相关的 API 函数，本小节以一个简单示例说明如何在服务器邮槽和客户端邮槽之间传输数据。服务器邮槽负责创建邮槽，并从邮槽中读取数据。

代码如下：

```
01  #include "stdafx.h"
02  #include <windows.h>
03  #define MAX_BUFFER_LEN  256                 //定义数据缓冲区的大小
04  int main(int argc, char* argv[])
05  {
06      HANDLE   hSlot;                         //定义服务器邮槽句柄
07      char     buffer[MAX_BUFFER_LEN];        //定义接收数据缓冲区
08      DWORD    nReadBytes;                    //定义存放读取数据个数的变量
09      hSlot = CreateMailslot("\\\\.\\Mailslot\\slotSample", 0,
10                             MAILSLOT_WAIT_FOREVER, NULL);
11      if (hSlot == INVALID_HANDLE_VALUE)
12      {
13          //循环从邮槽中读取数据，因为使用 MAILSLOT_WAIT_FOREVER，所以无限等待
14          printf("创建邮槽失败。错误代码=%d\n", GetLastError());
15          return 0;
16      }
17
18      while (ReadFile(hSlot, buffer, MAX_BUFFER_LEN,
19                      &nReadBytes, NULL) != 0)
20      {
21          printf("接收到邮槽数据=%.*s\n",  nReadBytes, buffer);
22      }
23      return 0;
24  }
```

上面代码首先调用 CreateMailslot()函数创建名称为 slotSample 的服务器邮槽，通过返回值判断创建邮槽是否成功。接着调用 ReadFile()函数，从邮槽中读取数据。因为在创建邮槽时使用的是 MAILSLOT_WAIT_FOREVER 参数，表示无时间限制地等待读取数据，因此，ReadFile 会一直等待读取数据。读取到数据后，向屏幕输出读取的数据。程序运行效果如图 17-1 所示。

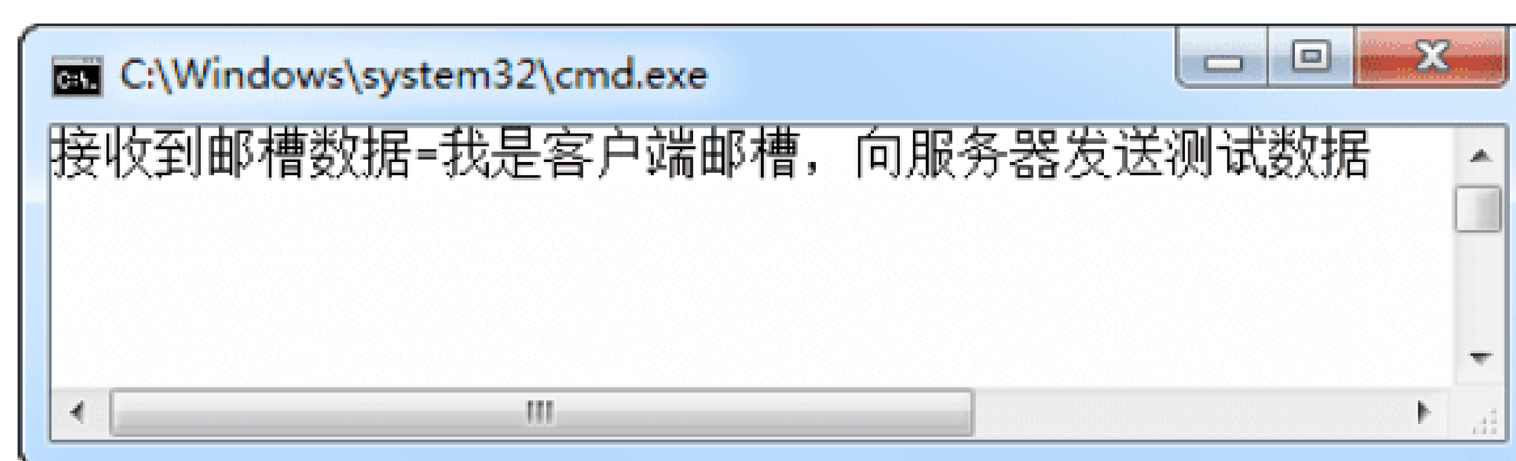


图 17-1　邮槽服务器运行效果

邮槽客户端的功能是向邮槽服务器发送数据。首先打开到服务器邮槽的连接，然后调用 WriteFile()函数向邮槽发送数据，服务器通过 ReadFile()就可以接收到客户端邮槽发送的数据。程序源代码如下：

```
01  #include <windows.h>
02  #include "process.h"
03
04  int main(int argc, char* argv[])
05  {
06      HANDLE      hSlot;                      //定义客户端邮槽句柄
07      DWORD       dwByteWrites;               //定义存放写入数据个数的变量
08      Char        ComputerName[256];
09      char        Content[256];               //定义消息内容
10      sprintf(ComputerName, "\\\\.\\Mailslot\\slotSample");
```

```
11
12      hSlot = CreateFile(ComputerName, GENERIC_WRITE, FILE_SHARE_READ,
13                  NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
14      if (hSlot == INVALID_HANDLE_VALUE)
15      {
16          printf("创建客户端邮槽失败。错误代码=%d\n", GetLastError());
17          return -1;
18      }
19
20      sprintf(Content, "我是客户端邮槽，向服务器发送测试数据");
21      if (WriteFile(hSlot, Content, strlen(Content),
22                  &dwByteWrites, NULL) == 0)
23      {
24          printf("向邮槽写入数据失败。错误代码=%d\n", GetLastError());
25          return -1;
26      }
27      printf("向邮槽写入%d个字节数据\n", dwByteWrites);
28      CloseHandle(hSlot);
29      system("pause");
30      return 0;
31  }
```

上面代码首先调用 CreateFile()函数打开指定名称的客户端邮槽，根据返回值判断是否打开客户端邮槽成功。如果成功，则调用 WriteFile()向邮槽发送数据，并向屏幕输出发送信息的情况。最后关闭客户端邮槽句柄。程序运行效果如图 17-2 所示。

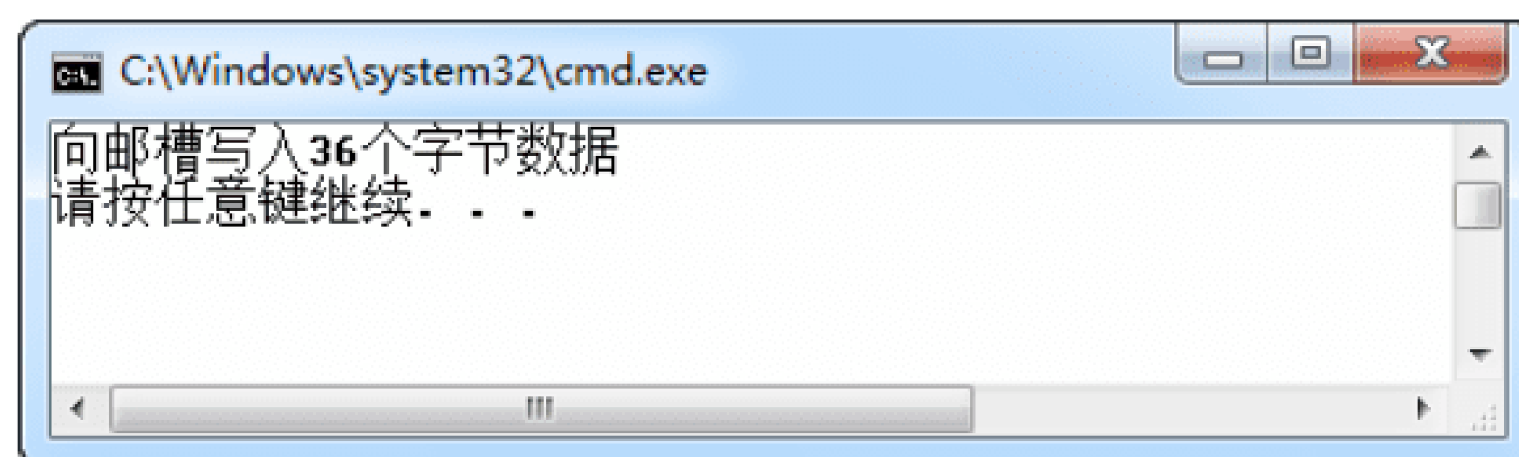


图 17-2　邮槽客户端运行效果

17.2　匿 名 管 道

17.1 节介绍了邮槽，而邮槽只支持网络的单向数据传输，要想双向传输就需要建立两个邮槽。为了简化操作，Win32 API 提供了只支持本地数据传输的匿名管道。这里所说的管道，是指具有对等两端的信息管道。Windows 支持单向数据传输的单向管道和双向数据传输的双向管道。

17.2.1　匿名管道的实施细节

匿名管道是未命名的本地管道，用于在父进程和子进程间传输数据。匿名管道不能在网络上进行数据传输。与邮槽不同的是，管道服务器可以为管道客户端设置读句柄和写句柄，从而使得管道客户端可以向管道服务器收发数据。使用匿名管道的步骤如下。

（1）使用 CreatePipe()函数创建匿名管道，其函数原型为：

```
BOOL CreatePipe(
    PHANDLE hReadPipe,                          //返回的匿名管道的读管道句柄
    PHANDLE hWritePipe,                         //返回的匿名管道的写管道句柄
    LPSECURITY_ATTRIBUTES lpPipeAttributes, //管道的安全属性设置
    DWORD nSize                                 //管道的大小
);
```

此函数返回两个句柄，一个是管道读句柄，一个是管道写句柄。读句柄对管道只有读权限，写句柄对管道只有写权限。要在管道中进行数据通信，管道服务器必须将读写句柄传入管道客户端进程中，这就需要通过继承管道实现，因此在安全属性中要设置管道的可继承性为 true。

（2）使用相关的方法向匿名管道客户端发送读写句柄。通常匿名管道用于将命令行程序的输入输出与 Windows 程序的对接。此时，需要在启动命令行程序前，设置标准输入和标准输出为匿名管道的写句柄和读句柄。

（3）调用 ReadFile()函数从管道读数据，调用 WriteFile()函数向管道中写数据。与文件不同的是，匿名管道不支持异步读写操作。

（4）操作完成后，调用 CloseHandle()函数关闭管道句柄。当进程终止时，与其相关的所有的管道句柄都会被关闭。

17.2.2　匿名管道应用示例

17.2.1 小节简单地介绍了匿名管道的实施细节，本小节以一个简单的示例，说明使用匿名管道如何实现进程间通信。在本例中，匿名管道客户端是一个命令行工具，实现的功能是：向“标准输出”和“标准错误日志”输出提示信息。其代码如下：

```
01  #include <iostream.h>
02  int main(int argc, char* argv[])
03  {
04      //向标准输出输出提示信息
05      cout << "这里是 PipeClientSample 的标准输出" << endl;
06      //向标准错误日志输出提示信息
07      cerr << "这里是 PipeClientSample 的错误输出" << endl;
08      return 0;
09  }
```

管道服务器的作用是创建匿名管道，调用管道客户端程序，将管道客户端程序的读写指向匿名管道的读写句柄。程序代码如下：

```
01  void CPipeServerSampleDlg::OnButtonConnect()
02  {
03      SECURITY_ATTRIBUTES sa;
04      HANDLE hRead,hWrite;
05      sa.nLength = sizeof(SECURITY_ATTRIBUTES);
06      sa.lpSecurityDescriptor = NULL;
07      sa.bInheritHandle = TRUE;
08
09      if (!CreatePipe(&hRead,&hWrite,&sa,0))
10      {
11          m_Log = "调用 CreatePipe 函数创建匿名管道失败";
12          UpdateData(FALSE);
13          return;
```

```
14      }
15
16      STARTUPINFO si;                         //定义启动结构
17      PROCESS_INFORMATION pi;                 //定义进程信息结构
18      si.cb = sizeof(STARTUPINFO);
19      GetStartupInfo(&si);                    //获取启动信息
20      si.hStdError = hWrite;
21      si.hStdOutput = hWrite;
22      si.wShowWindow = SW_HIDE;
23      si.dwFlags = STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;
24
25      if (!CreateProcess(NULL, "PipeClientSample", NULL,
26          NULL, TRUE, NULL, NULL, NULL, &si, &pi))
27      {
28          m_Log = "调用 CreateProcess 创建进程失败";
29          UpdateData(FALSE);
30          return;
31      }
32      CloseHandle(hWrite);
33
34      char buffer[4096] = {0};
35      DWORD dwByteReads;
36      while (true)
37      {
38          if (ReadFile(hRead, buffer, 4095, &dwByteReads, NULL) == NULL)
39              break;
40          m_Log += buffer;
41          UpdateData(FALSE);
42          Sleep(1000);
43      }
44  }
```

在匿名管道服务器中，会创建匿名管道。启动客户端程序进程，并设置客户端的标准输出和标准错误指向匿名管道的写操作。当启动客户端进程后，会将客户端程序的输出通过管道传输给管道服务器，管道服务器读取数据后，会将其显示在界面的日志文本框中。程序运行效果如图 17-3 所示。

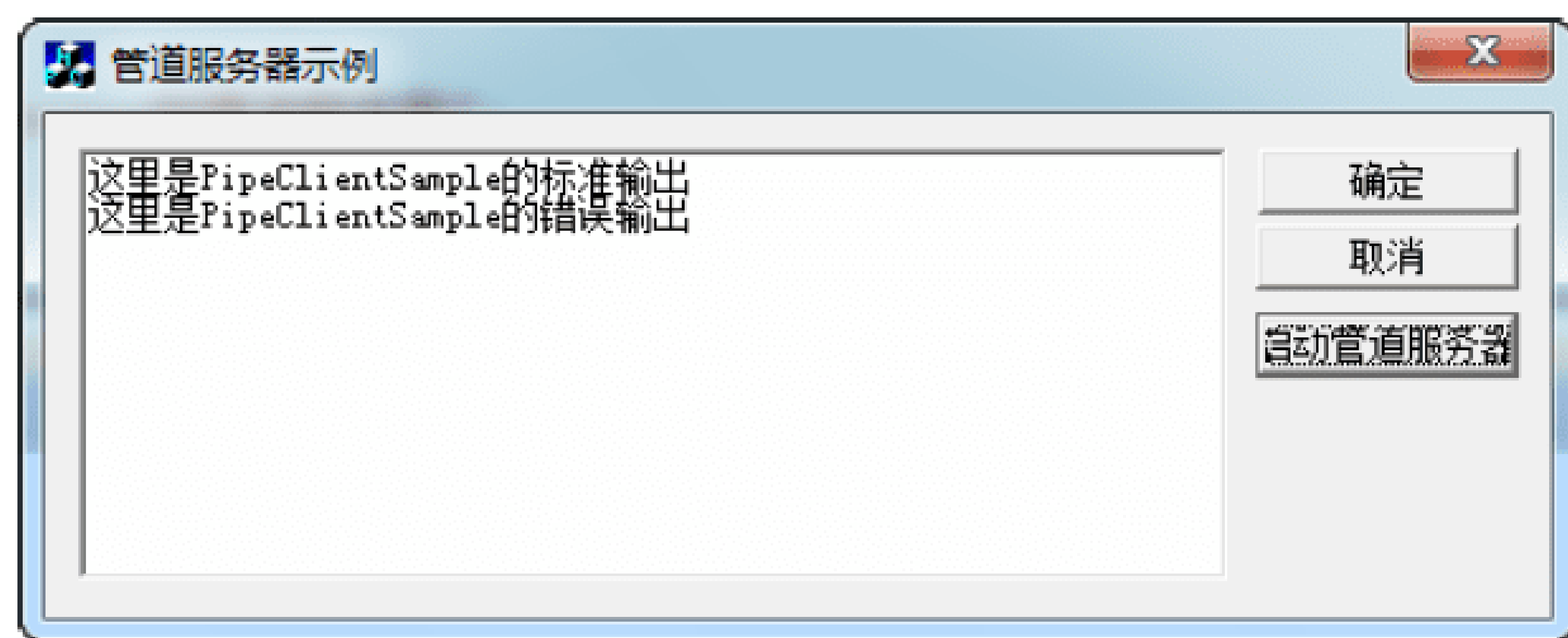


图 17-3　匿名管道示例运行效果

17.3　命名管道

虽然匿名管道简化了本地计算机上进程间的通信，但是因为没有唯一的名称来为管道命名，所以想要与其他进程进行数据通信时，无法指定固定的管道名称。鉴于此，Windows

提供了命名管道技术，为进程间的通信提供了更丰富的功能。本节将介绍有关命名管道的知识，并以一个示例讲解如何使用命名管道。

17.3.1　命名管道技术概述

命名管道可以看作邮槽的简单实现，用来实现进程间通信。命名管道是两个进程间交换数据的一种简单方式。命名管道的工作过程就像电话呼叫过程：每个命名管道只与一方通话，并且需要有建立管道连接和断开管道连接的过程。

命名管道是命名的单向地或双向地在管道服务器和管道客户端间进行通信的通道。命名管道的所有实例共享相同的管道名称，但是每个实例具有自己的数据缓冲区和句柄，并且客户端/服务器通信之间有单独的管道。任何进程都可以访问命名管道，使得命名管道可以简单地实现相关或非相关进程之间的通信。虽然进程既可以作为服务器，也可以作为客户端，但是在命名管道程序中，通常创建命名管道的进程称为管道服务器，连接命名管道的进程称为管道客户端。

17.3.2　命名规范及通信模式

每个命名管道具有唯一的名称，用于将其与系统中命名对象列表中的其他命名管道区分开。当管道服务器调用 CreateNamedPipe()函数创建命名管道时，需要指定创建的管道名称。当管道客户端调用 CreateFile()函数或 CallNamedPipe()函数连接命名管道时，也需要指定管道名称。其语法格式为：

```
\\计算机名称\pipe\PipeName
```

其中，计算机名称指定了命名管道所在的计算机在网络上使用的名称。PipeName 表示管道名称字符串，可以包括任何字符、数字和特殊字符。整个管道名称字符串不能超过 256 个字符，并且不区分大小写。因为管道服务器只能在本地计算机中的进程创建，因此，在 CreateNamedPipe()函数中，必须使用点号表示服务器名称，表示在本地创建命名管道，代码如下：

```
\\.\pipe\PipeName
```

最简单的命名管道通信模式是单服务器单客户端模式，此种模式下，服务器创建单个命名管道，连接到单个管道客户端，与客户端通信，从客户端处断开管道连接，关闭管道句柄，并终止管道服务器进程。

但是，最常见的是一个管道服务器与多个管道客户端通信的情况。此种方式下管道服务器可以使用单个管道实例与多个管道客户端相连，并可以按照顺序与每个客户端断开连接，但是此种方式性能会比较低。管道服务器应该创建多个管道实例提高同时处理与多个客户端的连接，这样程序性能会有所提高。一般有 3 种方式可以提高多管道程序的性能。

- ❑ 多线程管道服务器。此种方式每个实例句柄的线程只与单个管道客户端进行通信，虽然效率较高，但是系统根据需要为每个线程分配处理器时间，所以对于处理大量客户端的管道服务器，使用这种方式会降低程序性能，并且共享资源的同步也

会比较困难。

- 在 ReadFile()、WriteFile()和 ConnectNamedPipe()函数中指定 OVERLAPPED 结构，使用重叠操作。
- 使用 ReadFileEx()和 WriteFileEx()函数完成异步读写，在操作完成时，指定完成处理程序。

管道服务器可以同时存在多个管道实例，在第一次调用 CreateNamedPipe()函数时，使用 nMaxInstances 参数可以指定同时存在的管道实例的最大数量。服务器可以重复地调用 CreateNamedPipe()函数创建另外的管道实例，只要不超过实例的最大数量就可以。如果函数成功，则每次调用返回命名管道实例的服务器端的句柄。

一旦管道服务器创建了管道实例，管道客户端可以调用 CreateFile()函数或 CallNamedPipe()函数进行连接。如果管道实例可用，CreateFile()返回管道实例的客户端句柄。如果没有管道实例可用，管道客户端可以使用 WaitNamedPipe()函数等待管道服务器可用。

管道服务器可以通过调用 ConnectNamedPipe()函数决定何时管道客户端连接到管道实例。如果管道句柄在阻塞等待方式下，ConnectNamedPipe()直到有管道客户端连接时，才会返回。

17.3.3　使用命名管道

要使用命名管道，需要管道服务器与管道客户端之间的协作。其工作步骤如下。

（1）在命名管道服务器端调用 CreateNamedPipe()函数创建命名管道。在创建命名管道时，要注意字节管道类型和消息管道类型，分别用于发送字节流和消息单元。读者应该根据需要确定使用哪种类型的管道。

（2）服务器端调用 ConnectNamedPipe()函数，启动监听客户端的连接，从而允许命名管道的服务器进程等待客户端进程连接到命名管道实例。其函数原型为：

```
BOOL ConnectNamedPipe(
    //指向命名管道的服务器端句柄
    HANDLE hNamedPipe,
    //指向 OVERLAPPED 结构，用于设置重叠操作
    LPOVERLAPPED lpOverlapped );
```

命名管道服务器进程可以使用新创建的管道实例调用 ConnectNamedPipe()函数，也可以使用以前连接到其他客户端进程的实例，但是此时，需要先调用 DisconnectNamedPipe()函数断开与原来的客户端的连接，才可以与新客户端相连。否则如果以前的客户端关闭句柄，则 ConnectNamedPipe()函数返回 0，并且 GetLastError()函数返回 ERROR_NO_DATA，如果还没有关闭句柄，则返回 ERROR_PIPE_CONNECTED。

ConnectNamedPipe()函数有两种工作方式：阻塞和非阻塞。如果是阻塞方式，则函数直到有客户端连接过来时才返回；如果是非阻塞，则函数执行后立即返回。

（3）管道客户端使用 WaitNamedPipe()函数或 CallNamedPipe()函数连接命名管道。其中，CallNamedPipe()函数连接到消息类型管道，如果连接到字节类型的管道，则调用 CallNamedPipe()函数会失败。其函数原型为：

```
BOOL CallNamedPipe(
    LPCTSTR lpNamedPipeName,          //指向要连接的管道名称
```

```
    LPVOID lpInBuffer,                  //指向存放向管道写入数据的缓冲区指针
    DWORD nInBufferSize,                //指定输入缓冲区的大小
    LPVOID lpOutBuffer,                 //指向存放从管道读取数据的缓冲区指针
    DWORD nOutBufferSize,               //指定输出缓冲区的大小
    LPDWORD lpBytesRead,                //从管道中接收的要读取的字节数
    DWORD nTimeOut );                   //管道操作的超时时间
```

其中，nTimeOut 除了可以指定整型的毫秒数外，系统还预定了以下几个可用值。

- NMPWAIT_NOWAIT：不等待命名管道。如果命名管道不可用，函数返回错误。
- NMPWAIT_WAIT_FOREVER：无限期地等待。
- NMPWAIT_USE_DEFAULT_WAIT：使用在 CreateNamedPipe()函数中指定的默认值。

（4）接收到客户端的 WaitNamedPipe 请求后，服务器端会从 ConnectNamedPipe()函数中返回，以接收客户端的连接。

（5）客户端调用 CreateFile()函数打开到管道的连接。此时指定的文件访问参数需要与管道服务器中指定的参数兼容。与邮槽不同的是，命名管道支持异步读写模式，将 CreateFile()函数的 dwFlagsAndAttributes 参数设置为 FILE_FLAG_OVERLAPPED，即允许管道客户端以异步模式工作。

（6）在管道服务器进程和管道客户端进程之间，可以通过调用 ReadFile()函数和 WriteFile()函数进行双向的读写操作。

（7）调用 DisconnectNamedPipe()函数断开命名管道的连接后，调用 CloseHandle()函数关闭命名管道句柄。DisconnectNamedPipe()函数原型为：

```
BOOL DisconnectNamedPipe(
    HANDLE hNamedPipe );            //要断开的命名管道的实例句柄
```

如果调用此函数关闭命名管道，则再次访问此管道时将会发生错误。而且，管道中未读取的数据也会被删除。因此，在调用此函数断开连接前，应该调用 FlushFileBuffers()函数读取缓冲区中的所有数据。

17.3.4　其他功能函数

除了 17.3.3 小节介绍的使用命名管道的常用函数外，Windows 中还提供了其他与命名管道相关的 API。

GetNamedPipeHandleState()函数返回指定命名管道句柄的状态信息。这些信息在命名管道的有效期内，有时会发生变化。其函数原型为：

```
BOOL GetNamedPipeHandleState(
    HANDLE hNamedPipe,                  //要获取信息的命名管道句柄
    LPDWORD lpState,                    //句柄的当前状态
    LPDWORD lpCurInstances,             //存放当前管道实例数的变量的指针
    LPDWORD lpMaxCollectionCount,       //存放客户端发送前的最大字节数变量的指针
    LPDWORD lpCollectDataTimeout,       //在网络上传输数据的最大超时时间
    LPTSTR lpUserName,                  //存放客户端用户名变量的指针
    DWORD nMaxUserNameSize );           //获取用户名的长度
```

GetNamedPipeInfo()函数返回指定命名管道的信息。其函数原型为：

```
BOOL GetNamedPipeInfo(
    HANDLE hNamedPipe,                          //要获取信息的命名管道句柄
    LPDWORD lpFlags,                            //命名管道的类型
    LPDWORD lpOutBufferSize,                    //命名管道输出缓冲区的大小
    LPDWORD lpInBufferSize,                     //命名管道输入缓冲区的大小
    LPDWORD lpMaxInstances);                    //可以创建的最大管道实例数
```

PeekNamedPipe()函数从命名管道中复制数据到缓冲区中，但是不会删除管道中的数据，同时返回管道中的数据。函数原型为：

```
BOOL PeekNamedPipe(
    HANDLE hNamedPipe,                      //要获取信息的命名管道句柄
    LPVOID lpBuffer,                        //存放从管道中接收数据的缓冲区
    DWORD nBufferSize,                      //接收数据缓冲区的大小
    LPDWORD lpBytesRead,                    //表示从管道中读取的数据字节数
    LPDWORD lpTotalBytesAvail,              //表示管道中可读取的数据字节数
    LPDWORD lpBytesLeftThisMessage );       //表示消息中剩余的字节数
```

SetNamedPipeHandleState()函数可以设置管道句柄状态信息。其函数原型为：

```
BOOL SetNamedPipeHandleState(
    HANDLE hNamedPipe,                  //要设置信息的命名管道句柄
    LPDWORD lpMode,                     //要设置的新模式
    LPDWORD lpMaxCollectionCount,       //要设置的客户端发送数据前的最大字节数
    LPDWORD lpCollectDataTimeout );     //要设置的管道传输数据的最大超时值
```

其中，在 lpMode 参数中可以指定 PIPE_WAIT 或 PIPE_NOWAIT 改变管道句柄的等待方式。

17.3.5　命名管道实例

前面几小节介绍了命名管道的实现细节，本节以一个简单的示例演示如何使用命名管道实现进程间通信。命名管道的客户端程序的功能类似于 Socket 客户端的功能。调用 WaitNamedPipe()函数等待服务器命名管道创建成功后，调用 CreateFile()函数打开命名管道，通过 ReadFile()函数和 WriteFile()函数对命名管道进行读写操作。以下为命名管道客户端的代码。

```
01  void CNamedPipeClientSampleDlg::OnButtonConnect()
02  {
03      CString content = "{测试数据,我是命名管道客户端}\r\n"; //要发送的数据
04      DWORD dwWriteBytes;                                   //发送的数据长度
05      //等待与服务器的连接
06      if (WaitNamedPipe("\\\\.\\Pipe\\NamedPipeSample",
07                        NMPWAIT_WAIT_FOREVER) == FALSE)
08      {
09          m_Log = "等待连接失败!\r\n";                      //显示信息
10          UpdateData(FALSE);
11          return;
12      }
13      //打开已创建的管道句柄
14      HANDLE hPipe = CreateFile("\\\\.\\Pipe\\NamedPipeSample",
15                  GENERIC_READ | GENERIC_WRITE,0, NULL, OPEN_EXISTING,
```

```
16                        FILE_ATTRIBUTE_NORMAL, NULL);
17        if (hPipe == INVALID_HANDLE_VALUE)
18        {
19            m_Log += "打开命名管道失败!\r\n";                //记录日志信息
20            UpdateData(FALSE);
21            return;
22        }
23        else
24            m_Log += "打开命名管道成功!\r\n";                 //记录日志信息
25
26        if (!WriteFile(hPipe, content, content.GetLength(),
27                          &dwWriteBytes, NULL))             //向管道写入数据
28            m_Log +=  "向命名管道写入数据失败!\r\n";
29        else
30            m_Log +=  "向命名管道写入数据成功!\r\n";
31        UpdateData(FALSE);
32        CloseHandle(hPipe); //关闭管道句柄
33    }
```

上面代码在调用 WaitNamedPipe()函数和 CreateFile()函数打开到服务器命名管道之间的连接后，调用 WriteFile()函数向服务器命名管道发送测试数据，并调用 CloseHandle()函数关闭管道句柄。程序运行效果如图 17-4 所示。

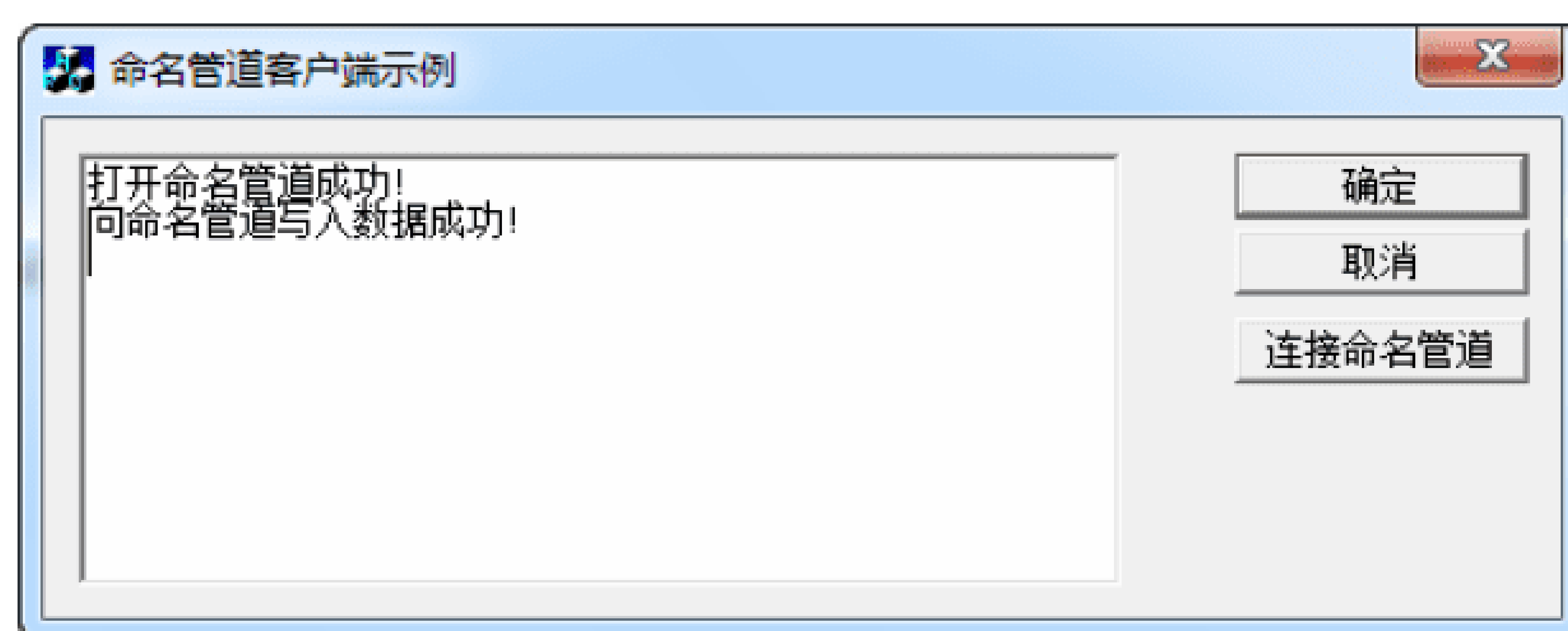


图 17-4　命名管道客户端运行效果

命名管道服务器程序功能类似于 Socket 服务器的功能，需要调用 CreateNamedPipe()函数创建命名管道，调用 ConnectNamedPipe()函数等待客户端管道连接。然后就可以调用 ReadFile()函数和 WriteFile()函数对命名管道进行读写。操作完成后，就可以调用 DisconnectNamedPipe()函数断开与客户端管道的连接。最后调用 CloseHandle()函数关闭服务器命名管道句柄，程序代码如下：

```
01  void CNamedPipeServerSampleDlg::OnButtonListen()
02  {
03      //创建命名管道
04      m_hPipe = CreateNamedPipe("\\\\.\\Pipe\\NamedPipeSample",
05          PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE,
06          1, 0, 0, 1000, NULL);
07      if (m_hPipe == INVALID_HANDLE_VALUE)    //判断命名管道是否成功
08      {
09          m_Log = "创建命名管道失败!\r\n";        //记录日志信息
10          UpdateData(FALSE);                  //显示日志
11          return;
12      }
13      else
14      {
15          m_Log = "创建命名管道成功!\r\n";        //记录日志信息
```

```
16          UpdateData(FALSE);                    //显示日志
17          AfxBeginThread(ListenProc, this);     //开启监听线程
18      }
19  }
20  //监听线程函数
21  UINT ListenProc(LPVOID lpVoid)
22  {
23      char buffer[1024];                        //数据缓存
24      DWORD dwReadBytes;                        //定义读取数据的字节个数的变量
25      //获取对话框句柄
26      CNamedPipeServerSampleDlg* pDlg =
27              (CNamedPipeServerSampleDlg*)lpVoid;
28      if (ConnectNamedPipe(pDlg->m_hPipe, NULL) == FALSE)
29      {
30          //等待客户端连接
31          CloseHandle(pDlg->m_hPipe);                     //关闭管道句柄
32         //记录日志信息
33          pDlg->m_Log += "与命名管道客户端建立连接失败!\r\n";
34          pDlg->m_editLog.SetWindowText(pDlg->m_Log); //显示日志
35          return 0;
36      }
37      else
38      {
39          pDlg->m_Log += "与客户端建立连接!\r\n";          //记录日志信息
40          pDlg->m_editLog.SetWindowText(pDlg->m_Log); //显示日志
41      }
42      if (ReadFile(pDlg->m_hPipe, buffer, sizeof(buffer),
43                  &dwReadBytes, NULL) == FALSE)
44      {
45          //从管道读取数据
46          CloseHandle(pDlg->m_hPipe);                     //关闭管道句柄
47          pDlg->m_Log +=  "从管道读取数据失败!\r\n";       //记录日志信息
48      }
49      else
50      {
51          buffer[dwReadBytes] = '\0';                     //显示接收到的信息
52          pDlg->m_Log += "接收到客户端命名管道发送的数据=\r\n";
53          pDlg->m_Log += CString(buffer);                 //记录日志信息
54      }
55
56      if (DisconnectNamedPipe(pDlg->m_hPipe) == FALSE)//终止连接
57      {
58          pDlg->m_Log += "终止连接失败!\r\n";             //记录日志信息
59      }
60      else
61      {
62          CloseHandle(pDlg->m_hPipe);                     //关闭管道句柄
63          pDlg->m_Log += "成功终止连接!\r\n";             //记录日志信息
64      }
65
66      pDlg->m editLog.SetWindowText(pDlg->m Log);     //显示日志
67      return 1;
68  }
```

在上面的代码中，服务器程序调用 CreateNamedPipe()函数创建完命名管道后，启动一个线程，此线程用于接收命名管道客户端的连接，并从客户端命名管道中读取数据。最后

断开与客户端命名管道的连接，并关闭服务器命名管道句柄，程序运行效果如图 17-5 所示。

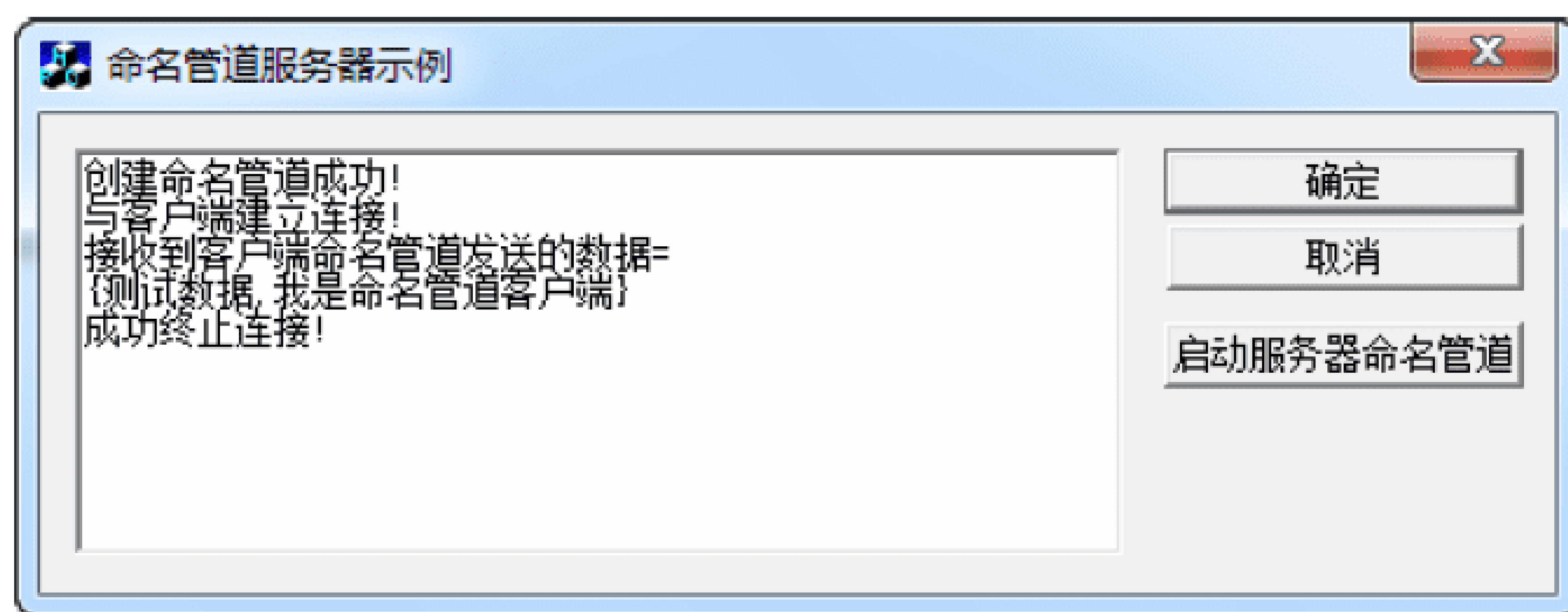


图 17-5　命名管道服务器运行效果

17.4　本 章 小 结

本章主要介绍了邮槽、匿名管道和命名管道的使用方法。这 3 种方式都是实现进程间通信的重要手段。邮槽是单向数据通信通道，管道是双向数据通信通道；匿名管道只支持本地进程间进行数据通信，命名管道支持网络进程间进行数据通信。本章的重点是掌握如何编写邮槽和管道程序。本章的难点是理解邮槽和管道的原理及实施细节。第 18 章将介绍通信端口的编程。

17.5　习　　题

创建基于对话框的应用程序，如图 17-6 和图 17-7 所示，分别作为服务器端和客户端。程序的使用方式如下。

（1）开启服务器，单击“创建邮槽”按钮。

（2）开启客户端，单击“连接邮槽”按钮。

（3）单击客户端的“写入数据”按钮，将客户端右边文本框中的内容写入到邮槽中。

（4）单击服务器端的“读取数据”按钮，将从邮槽中读取到的数据写入到服务器端右边的文本框中。

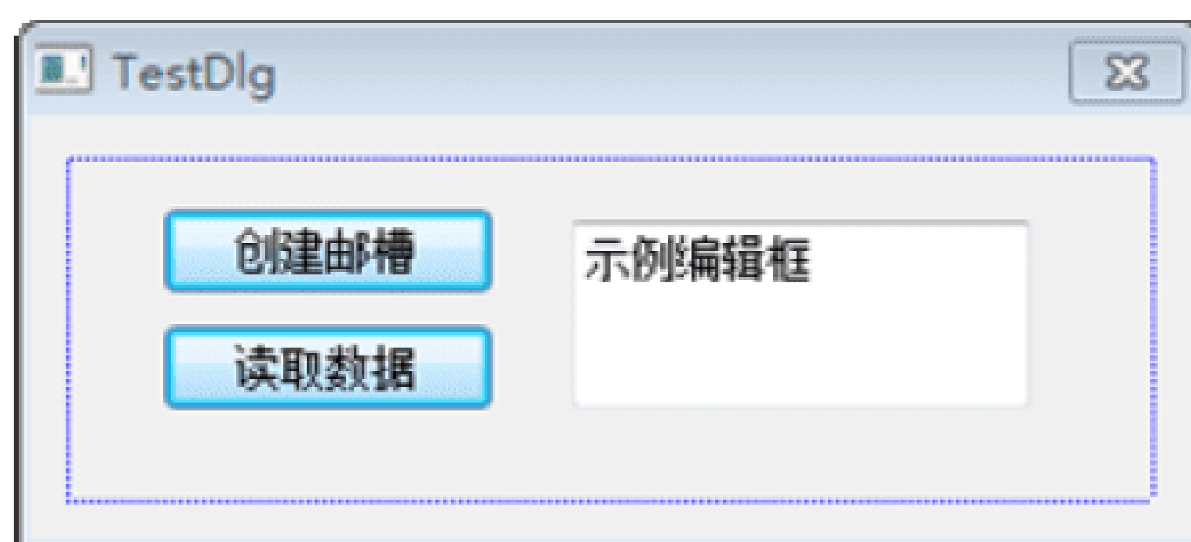


图 17-6　邮槽服务器端

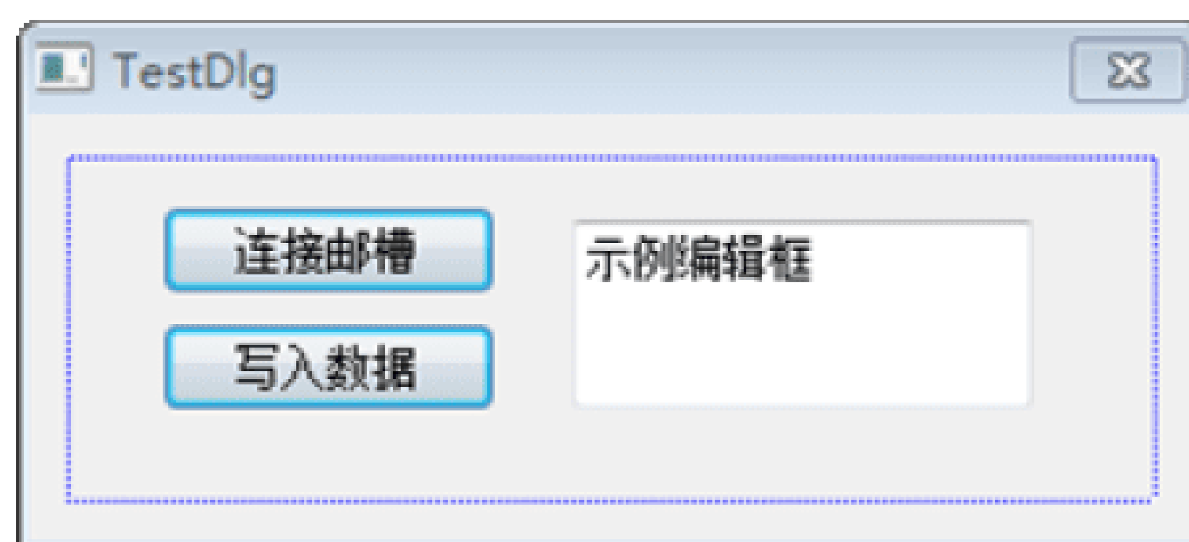


图 17-7　邮槽客户端

【思路】参考 17.1.5 小节的示例，在对话框的相应事件中填写创建邮槽、连接邮槽、向邮槽写入数据以及从邮槽中读取数据的代码。

第 18 章　通信端口编程

前面讲述了套接字和邮槽管道的开发，本章主要讲述计算机中通信端口的编程。在计算机中，主机与外设间传输数据的方式有两种，一种是通过串口进行通信，一种是通过并口进行通信。本章将讲述串行端口编程和并行端口编程的方法。

18.1　串行端口通信编程

在工业控制程序中，经常会遇到需要与串行端口进行通信的情况，此时就需要我们掌握串行端口通信编程的技术。本节主要讲解在 Windows 环境下进行串口编程的基本知识。通过本节的学习，读者应该能够掌握串口编程的实现步骤和注意事项，为实际串口应用打下基础。

18.1.1　Windows 环境下的串口编程

串行端口提供了计算机与外部串行设备之间的数据传输通道。现在每台计算机都提供了一个或多个串行端口，依次命名为 COM1、COM2 等，简称串口，可以连接鼠标、调制解调器、扫描仪和手机等。由于串行通信方便易行，所以应用广泛。串行端口是作为 CPU 和串行设备间的编码转换器。当数据从 CPU 经过串行端口发送出去时，字节数据被转换为串行的位。在接收数据时，串行的位将被转换为字节数据。应用程序要做的就是如何使用串口与串行终端设备进行数据传输。要与串口进行通信，不同平台下有不同的接口。

在 Windows 平台下，使用了通信驱动程序 Comm.drv，以便使用标准的 Windows API 函数发送和接收数据。驱动程序通常由串行设备制造商提供，以便将其硬件与 Windows 连接，如图 18-1 所示。

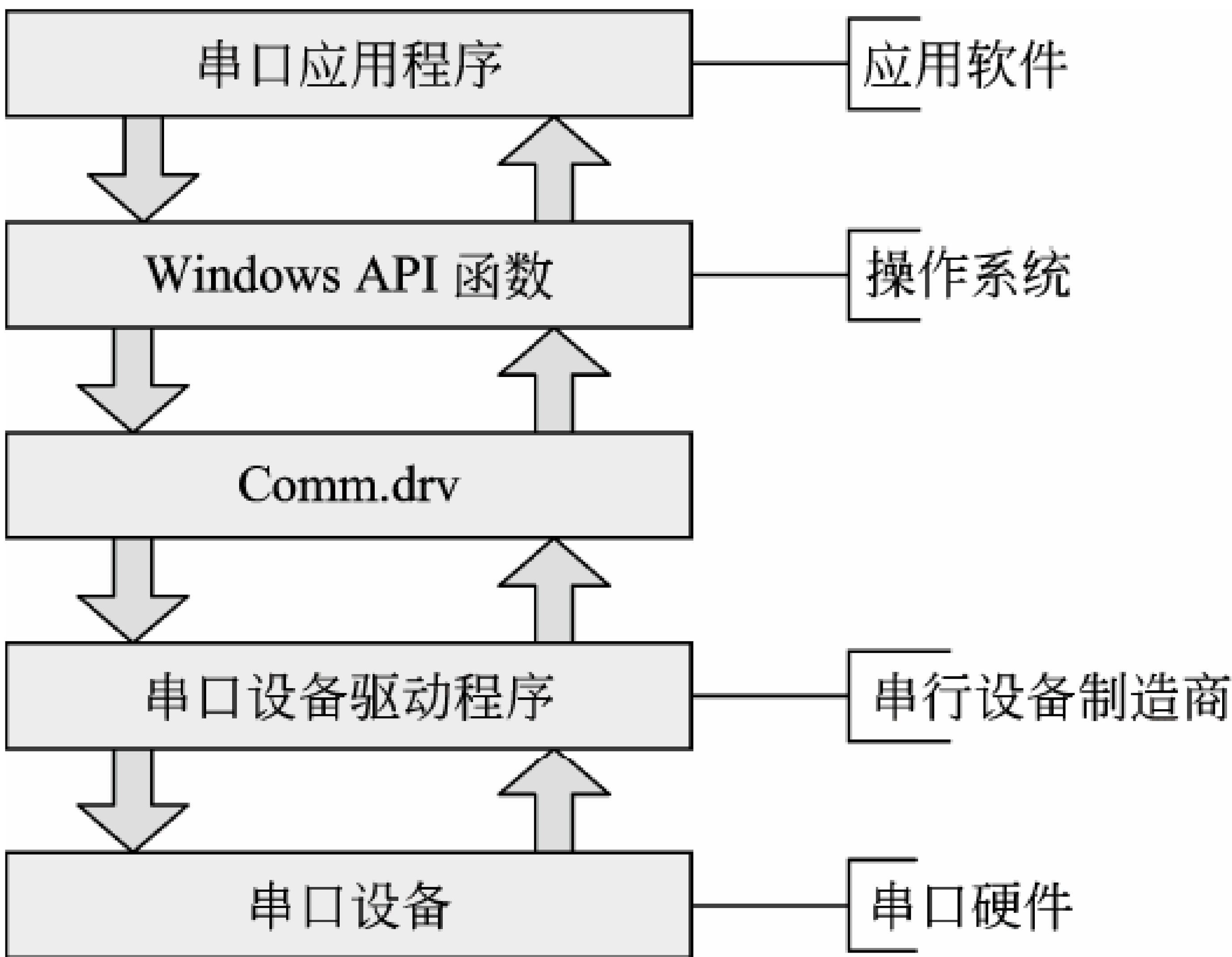


图 18-1　Windows 平台下串行端口通信实现流程

使用 Windows API 提供的通信函数可以编写出可移植的串行通信程序。在 Win16 中，使用专门为通信设备而提供的 OpenComm()、ReadComm()、WriteComm()、FlushComm() 和 CloseComm()函数实现对串口的操作。在 Win32 中，串口和其他通信设备被当作文件处理。所以，使用 CreateFile()函数打开串口，使用 CloseFile()函数关闭串口，使用 CommProp()函数、DCB 结构、GetCommProperties()函数、SetCommProperties()函数、GetCommState()函数及 SetCommState()函数等函数设置串口状态，使用 ReadFile()函数和 WritFile()函数读写串口。

在 Windows 平台下，还可以使用 VC 提供的串行通信控件 MSComm 与串口进行通信。MSComm 控件通过串行端口传输和接收数据，为应用程序提供串行通信功能。当然，MS Comm 也是通过调用 Windows API 的通信函数实现的，目的是将常用的串口功能封装在一起，提高开发效率。下面几个小节，将详细讲述如何使用这两种方式来操作串口。

技巧：深入理解 Comm.drv 对 Windows 平台下串口驱动程序的编写有很大帮助。

18.1.2　设定串口参数

使用串口进行数据通信，首先需要确认串口的工作模式，确定具体的工作参数，这样才可以使通信双方在约定好的工作模式下进行数据通信。如果参数设置得不正确，有可能使得传输出现错误，甚至死机。所以，在使用串口进行通信前，需要深入了解串口工作的具体过程和各个参数的详细含义。因此下面就具体讲述一下串口工作参数的结构和各个参数的具体用法。

在 VC 中，使用 DCB（Device-Control-Block，设备控制块）结构进行串口参数的配置。其结构定义如下：

```
typedef struct  DCB {             //DCB 结构
   DWORD DCBlength;       //DCB 结构的长度，单位是字节，赋值为 sizeof(dcb)
   DWORD BaudRate;        //波特率，具体为整数或 CBR_整数，如 300 与 CBR_300 等同
   DWORD fBinary: 1;            //是否是二进制模式，不进行 EOF 校验，必须为 true
   DWORD fParity: 1;            //是否进行奇偶校验
   DWORD fOutxCtsFlow:1;        //发送是否进行 CTS 控制
   DWORD fOutxDsrFlow:1;        //发送是否进行 DSR 控制
   DWORD fDtrControl:2;         //DTR 控制类型
   DWORD fDsrSensitivity:1;     //是否进行 DSR 信号处理
   DWORD fTXContinueOnXoff:1;   //接收 XOFF 后，传输是否继续
   DWORD fOutX: 1;              //发送是否进行 XON/XOFF 控制
   DWORD fInX: 1;               //接收是否进行 XON/XOFF 控制
   DWORD fErrorChar: 1;         //是否使用指定字符代替错误字节
   DWORD fNull: 1;              //是否丢掉 NULL 字符
   DWORD fRtsControl:2;         //RTS 控制类型
   DWORD fAbortOnError:1;       //发生错误时，是否终止读/写操作
   DWORD fDummy2:17;            //预留
   WORD wReserved;              //未使用
   WORD XonLim;                 //开始传输的最少字节数
   WORD XoffLim;                //停止传输的最多字节数
   BYTE ByteSize;               //每字节中的比特位数，范围是 4～8，数据传输时，
                                //每个字节不一定是 8 位
```

```
    BYTE Parity;                //奇偶校验方式 0～4 分别表示无校验，奇校验，偶校
                                //验，标识校验，空格校验
    BYTE StopBits;              //停止位 0、1、2 分别表示 1 个停止位、1.5 个停止
                                //位和 2 个停止位
    char XonChar;               //XON，标识开始的字符
    char XoffChar;              //XOFF，标识结束的字符
    char ErrorChar;             //发生错误时，替换错误字节的字符
    char EofChar;               //输入结束字符
    char EvtChar;               //接收的事件字符
    WORD wReserved1;            //预留；未使用
} DCB;
```

上面结构定义了串口工作所使用的所有参数。其中有几点需要注意。

（1）BaudRate 参数的波特率单位，与比特率是不同的概念。比特率是数字信号的传输速率，它用单位时间内传输的二进制代码的有效位（b）数表示，单位为每秒比特数 bit/s（bps）。波特率指数据信号对载波的调制速率，它用单位时间内载波调制状态改变次数表示，单位为波特（Baud）。

波特率与比特率的关系为：比特率=波特率×单个调制状态对应的二进制位数。波特率与比特率有如下的换算关系：1 Baud=log2M（b/s），其中 M 是信号的编码级数。一个信号往往可以携带多个二进制位，所以在固定的信息传输速率下，比特率往往大于波特率。换句话说，一个码元中可以传送多个比特。如 M=8，当波特率为 9600 时，数据传输率（比特率）为 28.8kb/s。因此，在编程时，注意确认正确的波特率。

（2）fParity 参数指定是否进行奇偶校验，Parity 参数表示使用的奇偶校验方式。奇偶校验就是指在传输数据时，在数据后加一位比特位（校验位），使得整个数据中（包括要传输的数据和增加的校验位）比特位为 1 的个数为奇数或偶数，分别称为奇校验和偶校验。具体校验方式如表 18-1 所示。

表 18-1　奇偶校验方式列表

常　　量	值	含　　义	描　　述	举　　例
EVENPARITY	2	偶校验	当进行奇偶校验时，使用偶校验，即校验位+数据位中，1 的个数为偶数个	数据：0110　1110；校验位：1
MARKPARITY	3	标志校验	始终设置校验位为 1	数据：0110　1110；校验位：1
NOPARITY	0	无奇偶校验	不进行奇偶校验	数据：0110　1110；校验位：无
ODDPARITY	1	奇校验	当进行奇偶校验时，使用奇校验，即校验位+数据位中，1 的个数为奇数个	数据：0110　1110；校验位：0
SPACEPARITY	4	空格校验	始终设置校验位为 0	数据：0110　1110；校验位：0

（3）StopBits 参数表示停止位比特数，其有效取值如表 18-2 所示。

表 18-2　停止位有效取值列表

常　　量	值	含　　义
ONESTOPBIT	0	1 个停止位
ONE5STOPBITS	1	1.5 个停止位
TWOSTOPBITS	2	2 个停止位

🔔技巧：在实际应用中，我们可以直接使用有效常量（如果存在）为每个参数赋值，也可以使用真实的值为其赋值。其中，true 可以使用 1 代替，false 可以使用 0 代替，但是建议尽可能使用常量来为参数赋值。

18.1.3　数据流控制参数

数据流控制是串口通信一个比较重要的特性。通过数据流控制程序可以控制数据发送和接收的步伐，以适应应用程序的需求。当然对数据流的控制是一个发送方和接收方双方之间的协作过程，要求双方互相配合、协调一致，否则会出现数据错误，甚至会出现程序死锁。所以，在使用数据流控制参数时，一定要理清各参数之间的协作关系。

fOutxCtsFlow 参数表示发送数据前，是否检测 CTS（clear-to-send，已清除完允许发送）信号，默认值为 true。如果此参数设置为 true，并且没有检测到 CTS，则发送将被挂起，直到检测到 CTS 信号。

fOutxDsrFlow 参数表示发送数据前，是否检测 DSR（data-set-ready，数据设备准备好）信号，默认值为 true。如果此参数设置为 true，并且没有检测到 DSR，则发送将被挂起，直到检测到 DSR 信号。

fDtrControl 参数表示 DTR（data-terminal-ready，数据终端准备好）流控制的工作方式。此参数的有效取值如表 18-3 所示。

表 18-3　fDtrControl的有效取值

常 量 名 称	含　　义
DTR_CONTROL_DISABLE	不使用 DTR 控制
DTR_CONTROL_ENABLE	使用 DTR 控制，进行数据传输前，只有检测到 DTR 信号，传输才会继续
DTR_CONTROL_HANDSHAKE	使用 DTR 硬握手。此情况下，不可以使用 EscapeCommFunction()函数

fDsrSensitivity 参数表示通信设备驱动是否处理 DSR 信号的状态。如果此参数设置为 true，并且 DSR 信号是非工作状态，则驱动将忽略此期间接收到的任何数据。

fTXContinueOnXoff 参数表示在输入缓冲区已满并且驱动程序已经发送 XoffChar 字符时，传输是否停止。如果此参数设置为 true，则当输入缓冲区中的字符达到 XoffLim 个，并且驱动程序已经发送 XoffChar 字符停止接收数据后，传输将继续；反之，直到输入缓冲区中空闲的空间达到 XonLim 个，并且驱动程序发送 XonChar 字符恢复接收数据后，传输才会继续。

fOutX 参数表示发送数据时，是否启用开/关数据流控制。如果此参数设置为 true，当接收到 XoffChar 定义的字符时，发送停止；当再次收到 XonChar 定义的字符时，重新开始发送数据。

fInX 参数表示接收数据时，是否启用开/关数据流控制。如果此参数设置为 true，当数据输入缓冲区中的字节数达到 XoffLim 时，发送 XoffChar 定义的字符；当数据输入缓冲区中空闲字节数达到 XonLim 时，发送 XonChar 定义的字符。此参数和上面的 fOutX 结合起来完成整个数据传输的开/关数据流控制，其工作过程如图 18-2 所示。

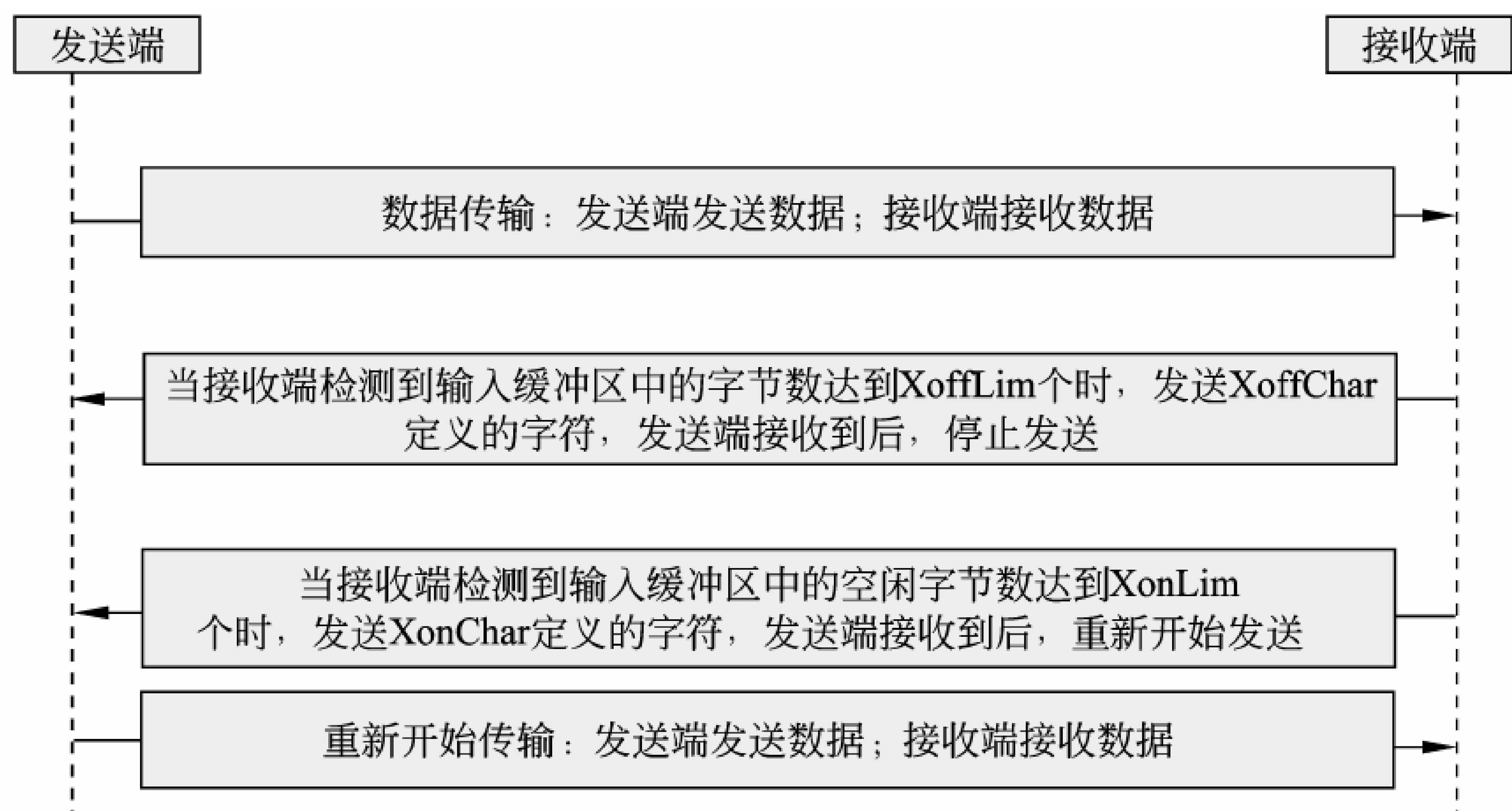


图 18-2　开/关数据流控制流程

fRtsControl 参数表示 RTS（request-to-send，请求发送数据）流控制的工作方式。此参数的有效取值如表 18-4 所示。

表 18-4　fRtsControl的有效取值

常 量 名 称	含　义
RTS_CONTROL_DISABLE	不使用 RTS 控制
RTS_CONTROL_ENABLE	使用 RTS 控制
RTS_CONTROL_HANDSHAKE	使用 RTS 硬握手。如果接收方的缓冲区中的数据少于缓冲区大小的一半，则发送 RTS 信号，直到其中的数据多于 3/4。此情况下，不可以使用 EscapeCommFunction()函数
RTS_CONTROL_TOGGLE	当有数据等待传输，则发送 RTS 信号，直到所有的数据传输完毕

fErrorChar 参数表示是否使用指定字符替换发生奇偶校验错误的字节。当此参数和 fParity 都设置为 true 时，驱动程序会用指定字符（ErrorChar 定义的字符）替换发生奇偶校验错误的字节。

fNull 参数表示当接收到 NULL 字节时，是否丢弃 NULL 字节，默认值为 true。如果此参数这时为 true，在接收到 NULL 字节时，则将其丢掉。但是建议在实际编程过程中，要注意自己程序的情况，结合程序的应用协议，因为很多协议中存在 NULL 字节的情况。这种情况下，如果此参数设置为 true，则可能导致程序无法正确工作。

fAbortOnError 参数表示当发生错误时，是否终止所有的读写操作，默认值为 true。如果此参数设置为 true，当发生错误时，则驱动程序会终止所有的读写操作，并产生一个错误报告。并且驱动程序在应用程序调用 ClearCommError()函数确认错误前将不再进行任何的通信操作。在实际应用中，通信编程的最难点就是错误处理，如果此参数设置为 true，则一定要处理好错误情况。

特殊字符参数主要定义了一些在传输过程中用到的，需要特殊处理的字符或参数值，如表 18-5 所示。

表 18-5　串口通信中的特殊字符参数

参　数	含　义
XonLim	表示发送 XonChar 字符前，输入缓冲区中最少的空闲字节数
XoffLim	表示发送 XoffChar 字符前，输入缓冲区中存储的最多的字节数

续表

参　　数	含　　义
XonChar	表示发送数据和接收数据过程中，开启数据传输的指示字符，与 fOutX 和 fInX 参数相关
XoffChar	表示发送数据和接收数据过程中，停止数据传输的指示字符，与 fOutX 和 fInX 参数相关
ErrorChar	表示接收数据时，用于替换发生奇偶校验错误字节的字符，与 fErrorChar 参数相关
EofChar	表示数据结束标识字符

18.1.4　申请串口资源

在了解了串口工作的参数后，就可以申请串口资源了。申请串口资源也就是连接串口，并打开串口。申请完串口资源后，还需要根据具体的需求结合对参数的了解，对串口进行参数配置。同时，还需要对串口工作的超时时间进行设置。具体步骤如下。

（1）调用 CreateFile()函数打开串口。在 Win32 中，将串口和管道等都作为一种特殊的文件对待。CreateFile()函数的定义如下：

```
HANDLE CreateFile(
  LPCTSTR lpFileName,                              //要打开的文件名称
  DWORD dwDesiredAccess,                           //访问方式（读-写）
  DWORD dwShareMode,                               //是否共享
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,      //安全属性
  DWORD dwCreationDisposition,                     //创建方式
  DWORD dwFlagsAndAttributes,                      //文件属性
  HANDLE hTemplateFile                             //模板文件句柄
);
```

使用 CreateFile()打开串口，要注意下面几点。

- ❑ dwShareMode 必须设置为 0，即串口在打开时，是不能共享的。
- ❑ dwCreationDisposition 必须设置为 OPEN_EXISTING，即串口只能打开已经存在的“文件”，而不能创建新的。
- ❑ hTemplateFile 必须设置为 NULL。
- ❑ dwFlagsAndAttributes 参数设置为 FILE_FLAG_OVERLAPPED，则串口以异步工作模式工作；如果不设置，则串口以同步工作模式工作。

下面的代码表示打开 COM1，并使用异步工作模式读写串口数据：

```
//打开串口 COM1
HANDLE hComm= CreateFile( "COM1",  GENERIC_READ | GENERIC_WRITE, 0, 0,
                      OPEN_EXISTING, FILE_FLAG_OVERLAPPED, 0);
if (hComm == INVALID_HANDLE_VALUE)
   {  //打开 COM1 错误，做错误处理  }
```

（2）使用 DCB 结构对串口参数进行设置，使用 GetCommState()函数或 BuildCommDCB()函数初始化 DCB 结构。其函数原型为：

```
BOOL GetCommState(
   HANDLE hFile,                        //通信设备句柄
   LPDCB lpDCB);                        //DCB 变量的指针
BOOL BuildCommDCB(LPCTSTR lpDef,        //要设置的参数字符串
   LPDCB lpDCB );                       //DCB 变量的指针
```

上面两个函数都会将当前 DCB 结构的值返回到 lpDCB 参数中。当打开串口时，系统自动使用最近一次使用该串口时使用的设置；当第一次使用该串口时，使用系统默认值。调用方式如下：

```
DCB dcb = {0};                                  //定义 DBC 结构变量
if (!GetCommState(hComm, &dcb))
{   //获得当前 DCB 设置失败   }
else
{   //获得当前 DCB 设置成功，可以继续进行   }
或
dcb.DCBlength = sizeof(dcb);
if (!BuildCommDCB("9600,n,8,1", &dcb))
{  //初始化 DCB 结构失败   }
else
{   //初始化 DCB 结构成功，可以继续  }
```

（3）根据需求修改 DCB 结构的各个参数值，调用 SetCommState()函数重新配置串口的参数。其函数定义为：

```
BOOL SetCommState(
    HANDLE hFile,                 //通信设备句柄
    LPDCB lpDCB);                 //DCB 变量的指针
```

下面为打开 COM1，设置波特率为 9600，停止位为 1，不进行奇偶校验的程序代码。

```
01  bool CComm19Dlg::OpenComm()                          //打开串口
02  {
03      HANDLE hComm= CreateFile( "COM1",  GENERIC_READ | GENERIC_WRITE,
04      0, 0,OPEN_EXISTING, FILE_FLAG_OVERLAPPED, 0);    //打开串口 COM1
05      if (hComm == INVALID_HANDLE_VALUE)
06          return false;                                //打开失败，则返回
07      DCB dcb;                                         //定义 DCB 结构
08      memset(&dcb, sizeof(dcb), 0);                    //初始化 DCB 结构
09      if (!GetCommState(hComm, &dcb))
10          return false;                                //获取当前的 DCB 取值
11      dcb.BaudRate = CBR_9600;                         //设置波特率
12      dcb.StopBits = 1;                                //设置数据停止位
13      dcb.Parity = NOPARITY;                           //设置奇偶校验方式
14      if (!SetCommState(hComm, &dcb))
15          return false;                                //设置新的串口参数
16      return true;
17  }
```

在编写串口通信程序时，参数设置尤为重要，这直接影响串口是否按照预想的工作方式工作，从而关乎着程序的正确性。所以在编写串口通信程序时，需要首先确认串口的工作方式，确定各个参数的取值。

技巧：在编写串口通信程序前，可以使用 Windows 自带的超级终端工具，连接串口看看串口是否工作正常，并可以与串口进行数据通信。这样，可以首先将各种协议数据在此工具下调通，减少调试代码的工作量。

（4）使用 COMMTIMEOUTS 结构和 SetCommTimeouts()函数设置串口工作的超时时间。其定义为：

```
typedef struct  COMMTIMEOUTS {
    DWORD ReadIntervalTimeout;          //两字符间传输的最大超时时间，单位是毫秒
    DWORD ReadTotalTimeoutMultiplier;   //读取每个字符时的超时时间值，单位是毫秒
    DWORD ReadTotalTimeoutConstant;     //每次读取时的超时时间值，单位是毫秒
    DWORD WriteTotalTimeoutMultiplier;  //写入每个字符时的超时时间值，单位是毫秒
    DWORD WriteTotalTimeoutConstant;    //每次写入时的超时时间值，单位是毫秒
} COMMTIMEOUTS,*LPCOMMTIMEOUTS;
```

在设置超时时间设置时，先调用 GetCommTimeouts()函数获得默认的超时时间设置，修改需要指定的超时设置后，再调用 SetCommTimeouts()函数修改。函数定义如下：

```
BOOL GetCommTimeouts(
    HANDLE hFile,                       //要获取的超时时间设置的串口句柄
    LPCOMMTIMEOUTS lpCommTimeouts);     //超时结构 COMMTIMEOUTS 变量
BOOL SetCommTimeouts(
    HANDLE hFile,                       //要设置的超时时间设置的串口句柄
    LPCOMMTIMEOUTS lpCommTimeouts );    //超时结构 COMMTIMEOUTS 变量
```

下面代码演示了如何设置串口的通信超时时间。

```
//定义 COMMTIMEOUTS 结构
COMMTIMEOUTS timeouts;
memset(&timeouts, sizeof(timeouts), 0);
//获取当前的 COMMTIMEOUTS 取值
if (!GetCommTimeouts(hComm, &timeouts))
    return false;
//设置读取两字符间的最大超时时间值
timeouts.ReadIntervalTimeout = 0;
//设置读取每个字符的超时时间值
timeouts.ReadTotalTimeoutMultiplier = 1000;
//设置每次读取的超时时间值
timeouts.ReadTotalTimeoutConstant = 5000;
//设置写入每个字符的超时时间值
timeouts.WriteTotalTimeoutMultiplier = 1000;
//设置每次写入的超时时间值
timeouts.WriteTotalTimeoutConstant = 5000;
//设置新的 COMMTIMEOUTS
if (!SetCommTimeouts(hComm, &timeouts))
    return false;
```

18.1.5　同步 I/O 读写数据

打开串口后，就可以对串口进行读写操作了。前面在讲述打开串口时，提到过同步工作模式和异步工作模式。本小节将讲述同步工作模式下的 I/O 读写操作。

同步 I/O 读写数据是指在进行读写操作时，函数直到读写操作完成后才返回。这种读写方式相对比较简单，但是需要注意对超时的控制和对资源的访问控制，避免发生死锁等情况。使用 ReadFile()函数可以从串口中读取数据，其函数原型为：

```
BOOL ReadFile(
    HANDLE hFile,                 //要读取数据的串口资源句柄的 GENERIC READ 权限
    LPVOID lpBuffer,                  //指向接收读取的数据的缓冲区的指针
    DWORD nNumberOfBytesToRead,       //指定要读取的数据的个数
```

```
    LPDWORD lpNumberOfBytesRead,     //指向存放成功读取的字节数的变量指针
    LPOVERLAPPED lpOverlapped );     //指向异步方式读取数据的 OVERLAPPED 结构
```

使用 WriteFile()函数可以向串口发送数据，其函数原型为：

```
BOOL WriteFile(
    HANDLE hFile,                //要发送数据的串口的句柄 GENERIC_WRITE 选项
    LPCVOID lpBuffer,            //指向包含要写入的数据的缓冲区的指针
    DWORD nNumberOfBytesToWrite,//指定要通过串口发送的数据的字节数
    LPDWORD lpNumberOfBytesWritten,//指向存放函数实际发送字节数的变量
    LPOVERLAPPED lpOverlapped );//指向异步方式读写数据的 OVERLAPPED 结构
```

从上面可以看出，从串口同步读写数据与从文件同步读写数据的方式是相同的。这也是 Win32 在原有的 API 函数基础上做的改进。

18.1.6　使用事件驱动机制

进程通过监控在通信资源中发生的一组事件处理串口通信。如应用程序可以使用事件监控 CTS 和 DSR 信号的状态改变，确定何时发送数据和何时接收数据。使用 SetCommMask()函数可以注册监控的事件。其函数原型为：

```
BOOL SetCommMask(
     HANDLE hFile,             //指定串口设备句柄，是由 CreateFile()函数返回的句柄
     DWORD dwEvtMask );        //指定要监控的事件，0 表示关闭对所有事件的监控
```

串口支持的事件如表 18-6 所示。

表 18-6　串口事件

事　件　值	含　　义
EV_BREAK	输入中断事件
EV_CTS	CTS 信号改变时的事件
EV_DSR	DSR 信号改变时的事件
EV_ERR	行状态发生错误时的事件，包括 CE_FRAME、CE_OVERRUN 和 CE_RXPARITY
EV_RING	检测到振铃
EV_RLSD	RLSD 信号改变时的事件
EV_RXCHAR	接收到字符，并将其放置到输入缓冲区中时的事件
EV_RXFLAG	接收到事件字符，并将其放置到输入缓冲区中的事件。事件字符在设备的 DCB 结构中指定，使用 SetCommState()函数设置
EV_TXEMPTY	输出缓冲区中的最后一个字符发送完成时的事件

进程使用 GetCommMask()函数可以获取串口注册的监控事件。其函数原型为：

```
BOOL GetCommMask(
    HANDLE hFile,              //指定串口设备句柄，是由 CreateFile()函数返回的句柄
    LPDWORD lpEvtMask);        //存储了注册的事件的值
```

应用程序指定要监控的事件后，进程使用 WaitCommEvent()函数可以等待一个或多个事件发生。可以用在同步或重叠操作中。当注册的事件发生时，进程完成等待操作，并设置事件变量表示检测到的事件类型。但是当调用 SetCommMask()函数时，WaitCommEvent()函数返回错误。WaitCommEvent()函数检测从最近一次调用 SetCommMask()函数或

WaitCommEvent()函数之后发生的注册事件。应用程序应该在程序初始化时，指定要监控的事件，并编写事件发生时执行的处理代码。以下是使用事件驱动机制的代码。

```
01  HANDLE hCom;                                            //串口句柄
02  OVERLAPPED o;                                           //OVERLAPPED 结构
03  BOOL fSuccess;                                          //操作是否成功
04  DWORD dwEvtMask;                                        //事件位
05  hCom = CreateFile( "COM1", GENERIC READ | GENERIC WRITE, 0,
06   NULL, OPEN_ EXISTING,FILE_FLAG_OVERLAPPED,  NULL );//打开串口 COM1
07  if (hCom == INVALID_HANDLE_VALUE)
08      {  处理错误 }
09  fSuccess = SetCommMask(hCom, EV_CTS | EV_DSR);      //注册事件
10  if (!fSuccess) { 错误处理}
11  //创建事件，等待事件对象
12  o.hEvent = CreateEvent( NULL, false, false, NULL);
13  //等待注册的串口事件发生
14  if (WaitCommEvent(hCom, &dwEvtMask, &o))
15  {
16      if (dwEvtMask & EV_DSR)
17      {
18           //检测到 DSR 事件后执行的代码
19      }
20      if (dwEvtMask & EV CTS)
21      {
22           //检测到 CTS 事件后执行的代码
23      }
24  }
```

上面代码注册了 DSR 事件和 CTS 事件，并检测事件的发生。在检测到事件后，判断其类型，并进行相应的处理。

18.1.7　异步 I/O 读写数据

要执行串口的异步 I/O 读写数据的功能，可以通过 18.1.5 小节中介绍的 ReadFile()函数和 WriteFile()函数实现。但是，本小节介绍两个专门用于异步的读写函数。使用 ReadFileEx()函数可以实现串口异步读数据操作，即当从串口读数据时，允许应用程序执行其他操作，并且能报告异步完成状态。当读操作完成或取消时，会调用指定处理代码。其函数原型为：

```
BOOL ReadFileEx(
   HANDLE hFile,            //指定串口设备句柄，具有 FILE FLAG OVERLAPPED 权限
   LPVOID lpBuffer,                   //指向接收读取的数据的缓冲区的指针
   DWORD nNumberOfBytesToRead,        //指定要读取的数据的个数
   LPOVERLAPPED lpOverlapped,         //异步操作结构 OVERLAPPED
   //读操作完成或取消执行的函数
   LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine );
```

如果 ReadFileEx()函数读取数据成功，输入缓冲区中所有数据读完，则等待状态返回 WAIT_IO_COMPLETION。如果函数执行成功，输入缓冲区中还有数据未读完，则返回 ERROR_IO_PENDING。要取消未完成的异步 I/O 操作，可以使用 CancelIO()函数。其中 OVERLAPPED 结构包含执行异步操作的输入和输出信息，可使用 HasOverlappedIoCompleted 宏确定异步 I/O 操作是否完成。其定义如下：

```
typedef struct  OVERLAPPED {
    DWORD  Internal;                    //指定依赖于系统的状态值
    DWORD  InternalHigh;                //指定数据传输的长度
    DWORD  Offset;                      //指定开始读取数据的偏移量
    DWORD  OffsetHigh;                  //从串口读数据时无效
    HANDLE hEvent; } OVERLAPPED;        //当数据传输完成时，设置信号的事件句柄
```

使用 WriteFileEx()函数可以实现串口异步写数据操作，即当向串口写数据时，允许应用程序执行其他操作，并且其能报告异步完成状态。当写操作完成或取消时，会调用指定的处理函数。其函数原型为：

```
BOOL WriteFileEx(
    HANDLE hFile,          //指定串口设备句柄，具有 FILE FLAG OVERLAPPED 权限
    LPCVOID lpBuffer,                   //指向发送数据的缓冲区的指针
    DWORD nNumberOfBytesToWrite,        //指定要发送数据的个数
    LPOVERLAPPED lpOverlapped,          //异步操作结构 OVERLAPPED
    //写操作完成或取消执行的函数
    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine );
```

如果 WriteFileEx()函数写数据成功，输出缓冲区中所有数据发送完，则等待状态返回 WAIT_IO_COMPLETION。如果函数执行成功，输出缓冲区中还有数据未发送完，则返回 ERROR_IO_PENDING。要取消未完成的异步 I/O 操作，可以使用 CancelIO()函数。使用 ReadFileEx()函数和 WriteFileEx()函数可以完成异步 I/O 读写数据的功能，在执行串口读写操作时不影响其他处理过程，从而可以开发出具有并行处理能力的程序。

18.1.8　MS Comm 串行通信控件

为了简化串口编程，微软提供了一个基于串口通信的 COM 组件——MS Comm 串行通信控件。本小节介绍如何使用此控件进行串行通信的编程。其步骤如下。

（1）按照前面讲过的方法创建对话框应用程序。

（2）添加 MS Comm 组件。右击对话框资源，在弹出的快捷菜单中选择“插入 ActiveX 控件”命令，打开“插入 ActiveX 控件”对话框。选择 Microsoft Communications Controls, version 6.0 选项，单击“确定”按钮。

说明： Visual C++ 6.0 在安装时会自动添加并注册这个组件，但是 Visual Studio 2010 在安装时没有这么做，所以要想在 Visual Studio 2010 中使用这个组件就需要读者自己来注册这个组件，方法如下。

① 取得组件文件。装过 Visual C++ 6.0 的系统，可以在系统盘/windows/system32 下找到 MSCOMM32.OCX、MSCOMM32.SRG 和 MSCOMM32.DEP 这 3 个文件。读者要把它们复制到自己的系统盘的相同目录下。

② 注册这个组件。在命令行中使用命令 regsvr32 MSCOMM32.OCX。

③ 修改注册表。将以下内容写到文本文件中，修改扩展名为.reg，然后双击这个文件名即可。

```
REGEDIT
```

```
HKEY CLASSES ROOT/Licenses=Licensing
HKEY CLASSES ROOT/Licenses/4250E830-6AC2-11cf-8ADB-00AA00C00905        =
kjljvjjjoquqmjjjvpqqkqmqykypoqjquoun
```

（3）在对话框资源中添加控件，包括 1 个发送编辑框、1 个接收编辑框、1 个日志编辑框、1 个打开按钮和 1 个发送按钮。

（4）在对话框类中添加以下代码：

```
01  //打开串口
02  void CMSCOMMSampleDlg::OnButtonOpen()
03  {
04      if(m_Comm.GetPortOpen())          //如果串口是打开的，则先关闭串口
05          m Comm.SetPortOpen(FALSE);
06
07      m_Comm.SetCommPort(1);            //选择 COM1
08      m_Comm.SetInBufferSize(1024);     //接收缓冲区
09      m_Comm.SetOutBufferSize(1024);    //发送缓冲区
10      //设置当前接收区数据长度为 0,表示全部读取
11      m Comm.SetInputLen(0);
12      m_Comm.SetInputMode(0);           //以文本方式读写数据
13      //接收缓冲区有 1 个及 1 个以上字符时，触发 OnComm 事件
14      m Comm.SetRThreshold(1);
15      //波特率 9600 无检验位，8 个数据位，1 个停止位
16      m Comm.SetSettings("9600,n,8,1");
17
18      if(!m_Comm.GetPortOpen())         //如果串口没有打开则打开
19          m_Comm.SetPortOpen(TRUE);     //打开串口
20      else
21          m Comm.SetOutBufferCount(0);
22      WriteLog("打开串口成功");
23  }
```

上面代码使用 SetCommPort 设置打开的串口的编号，使用 SetInBufferSize()函数设置输入缓冲区的大小，使用 SetOutBufferSize()函数设置输出缓冲区的大小，使用 SetInputLen()设置接收区数据的长度，使用 SetInputMode()函数设置读写数据的方式是文本方式还是二进制方式，使用 SetRThreshold()函数设置有字符到达时触发 OnComm 事件，使用 SetSettings()函数设置串口的配置参数，最后调用 SetPortOpen()函数打开串口。下面是事件处理函数的代码。

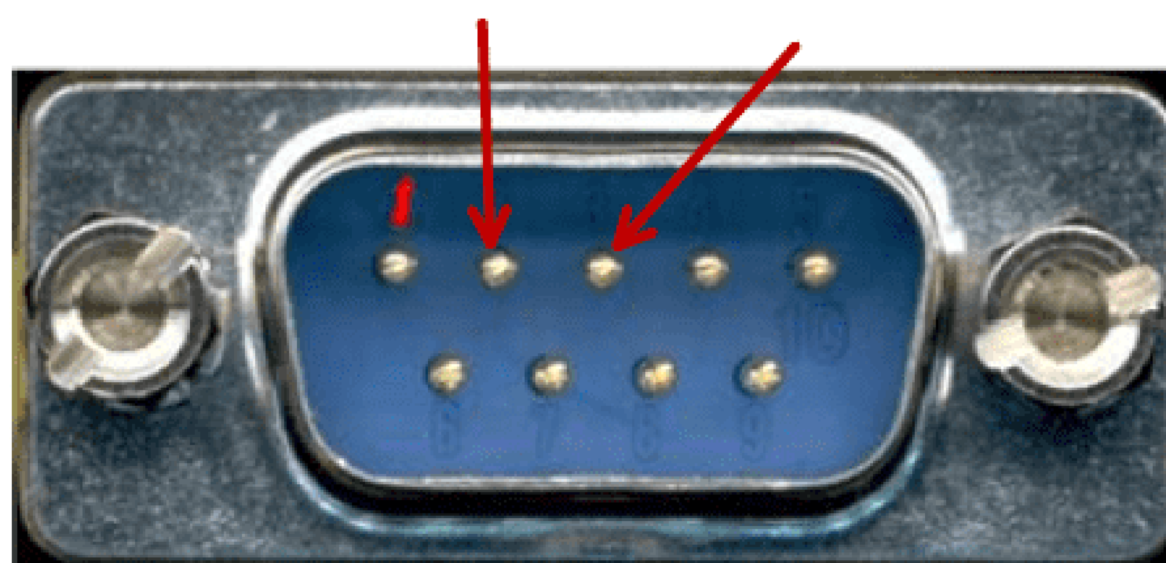
```
01  //事件处理函数
02  void CMSCOMMSampleDlg::OnOnCommMscomm1()
03  {
04      VARIANT vInput;                          //输入变量
05      COleSafeArray arrInput;                  //输入数组
06      LONG len,k;
07      BYTE rxdata[2048];                       //设置 BYTE 数组
08      CString strtemp;
09      switch(m Comm.GetCommEvent())
10      {
11      case EV_CTS:                             //发送数据
12          WriteLog("发送数据");
```

```
13          break;
14      case EV_RXCHAR:                              //读取数据
15          vInput=m_Comm.GetInput();                //读取缓冲区
16          arrInput=vInput;
17          len=arrInput.GetOneDimSize();            //得到有效数据长度
18          for(k=0; k < len; k++)                   //读取数据
19          {
20              arrInput.GetElement(&k,rxdata+k);//转换为 BYTE 型数组
21              BYTE bt=*(char*)(rxdata+k);          //字符型
22              //将字符送入临时变量 strtemp 中存放
23              strtemp.Format("%c",bt);
24              m_editReceive+=strtemp;
25          }
26          WriteLog("接收数据");
27          break;
28      default:                                     //出错
29          m_Comm.SetOutBufferCount(0);
30          break;
31      }
32      UpdateData(FALSE);                           //更新文本框内容
33      return ;
34  }
```

上面代码定义的 OnOnCommMscomm1()函数用于处理串口事件，GetCommEvent()函数可以获取当前串口控件的事件。当事件为 EV_CTS 时，表示准备发送数据；当事件为 EV_RXCHAR 时，表示有新数据到达。下面代码演示了当用户单击“发送”按钮时执行的函数。

```
01  void CMSCOMMSampleDlg::OnButtonSend()
02  {
03      UpdateData(TRUE);
04      unsigned char  uiSum=0;
05      int iLen = m_editSend.GetLength();
06      CByteArray  array;
07
08      for(int i=0; i<iLen-1; i++)
09          uiSum+=m_editSend[i];                        //计算校验和
10      array.RemoveAll();                               //清空数组
11      array.SetSize(iLen+1);                           //设置数组大小为帧长度
12      for(int i=0; i<iLen; i++)                        //把待发送数据存入数组
13          array.SetAt(i,m_editSend[i]);
14      array.Add(uiSum);                                //加入校验码
15
16      if (!m_Comm.GetPortOpen())
17          m_Comm.SetPortOpen(TRUE);
18
19      m_Comm.SetOutput(COleVariant(array));
20  }
```

上面代码使用组件的 SetOutput()函数将要发送的数据放置到发送缓冲区中。图 18-3 所示为 MS Comm 控件实例的运行效果图，单击“打开”按钮后，在发送编辑框中输入要发送的数据，单击“发送”按钮，则会在接收编辑框中显示接收到的数据。注意需要将串口的发送脚和接收脚连接起来。



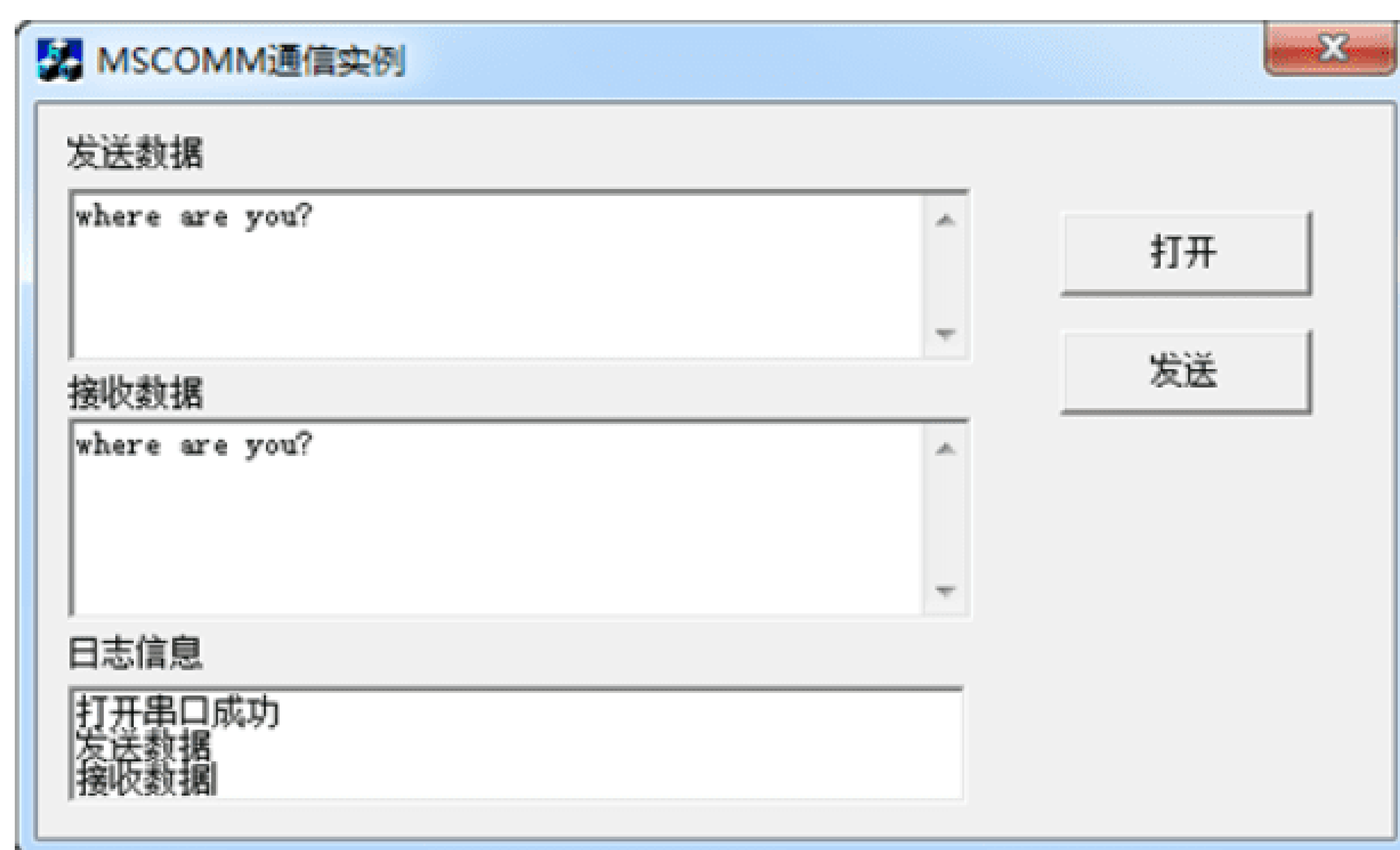


图 18-3　MS Comm 控件实例运行效果

18.2　通信端口编程实例

本节介绍如何使用 Win32 API 函数进行通信端口的编程。因为串口数据接收需要不停地监视串口和处理串口事件，因此，在本实例中，将其封装为单个线程执行类，有关线程编程的知识在后面的章节中会详细介绍。本节将着重讲解对串口的操作。

18.2.1　串口线程初始化

串口线程初始化由 CThreadCom 对象的构造函数完成。它实现串口工作变量的初始化工作。代码如下：

```
01  //串口线程类的构造函数
02  CThreadCom::CThreadCom(HANDLE hCom)
03  {
04      m_hCom=hCom;                       //初始化串口句柄
05      m_bInit=false;                     //串口初始化状态为 false
06      m_sCom="";                         //清空串口名称
07      m_sError="No Error!";              //初始化错误信息
08      m_hThread=NULL;                    //置空线程句柄
09      m_dwSendMsgToParent=0;             //初始化接收“发送”消息的句柄
10      m_dwRecvMsgToParent=0;             //初始化接收“接收”消息的句柄
11      m_pWndParent=NULL;                 //初始化父窗口的句柄
12      memset((unsigned char*)&m_overRead,0,sizeof(OVERLAPPED));
13                                         //初始化读异步变量
14      memset((unsigned char*)&m_overWrite,0,sizeof(OVERLAPPED));
```

```
15                                              //初始化写异步变量
16      m_overRead.hEvent=CreateEvent(NULL,true,false,NULL);//创建读事件
17      m_overWrite.hEvent=CreateEvent(NULL,true,false,NULL);//创建写事件
18  }
19  CThreadCom::~CThreadCom()                   //串口线程类的析构函数
20  {
21      CloseHandle(m_overRead.hEvent);         //关闭读异步事件
22      CloseHandle(m_overWrite.hEvent);        //关闭写异步事件
23  }
```

上面代码是串口线程类的构造函数和析构函数。构造函数初始化类中的变量，尤其是异步结构 OVERLAPPED，并创建事件。在析构函数中，关闭读写异步结构中的事件。因为串口线程类是继承自 CWinThread 类，所以需要重载 CWinThread 的几个函数。代码如下：

```
01  //重载线程类的 InitInstance()初始化实例函数
02  BOOL CThreadCom::InitInstance()
03  {
04      m_bAutoDelete=false;                    //初始化是否自动删除变量为 false
05      m_bDone=false;                          //初始化是否完成变量为 false
06      return true;                            //返回 true
07  }
08  //重载线程类的 ExitInstance()退出实例函数
09  int CThreadCom::ExitInstance()
10  {
11      BOOL bFlag=CloseCom();                  //关闭串口
12      return CWinThread::ExitInstance();      //调用基类的 ExitInstance()函数
13  }
```

18.2.2　串口接收线程

串口接收线程的处理工作由 CThreadCom 对象的 Run()函数完成。它实现循环从串口读取数据，并将接收到的数据发送给处理函数的功能。代码如下：

```
01  int CThreadCom::Run()                       //重载线程类的 Run()运行函数
02  {
03      DWORD dwError,dwReadNum,dwByteRead,dwEvent;//定义需要的变量
04      COMSTAT ComStat;                        //定义串口状态变量
05      BYTE rBuf[MAXCOMINBUF];                 //定义输入数据缓冲区
06      while (!m_bDone)                        //根据是否自动结束变量进行处理循环
07      {
08          while (m_hCom!=INVALID_HANDLE_VALUE)//如果串口句柄有效
09          {
10              if(::WaitCommEvent(m_hCom,&dwEvent,NULL))
11                                                  //等待注册的串口事件发生
12              {
13                  dwByteRead=0;                   //初始化读取的字节数为 0
14                  if((dwEvent & EV_RXCHAR))       //如果有数据到达
15                  {
16                      ClearCommError(m_hCom,&dwError,&ComStat);
17                                                  //清空当前串口事件
18                      if(ComStat.cbInQue!=0)//如果输入队列中的数据个数不为 0
19                      {
20                          dwReadNum=ComStat.cbInQue;//设置读取的数据个数
```

```
21                          dwByteRead=0;               //初始化读取的字节数为 0
22                          if(dwReadNum>200) dwReadNum=200;
23                                                      //每次最多读取 200 个
24                          memset(rBuf,0,sizeof(rBuf));//清空接收缓冲区变
25                                                           //量
26                          DWORD i=::ReadFile(m_hCom,rBuf,dwReadNum,
27                              &dwByteRead,&m_overRead);//读取数据
28                      for(i=dwByteRead;i<1024;i++) rBuf[i]=0;
29                                                      //使用 0x0 填充剩余字节
30                  }
31              }
32          if(dwByteRead) if(m_pWndParent)//如果读取数据，则向父窗口发送消息
33              m_pWndParent->SendMessage(m_dwRecvMsgToParent,
34                                      (DWORD)rBuf,dwByteRead);
35              }
36          }
37          Sleep(1000);                        //线程休眠
38      }
39      return CWinThread::Run();               //调用基类的 Run()函数
40  }
```

上面代码分别重载了初始化实例 InitInstance()函数、退出实例 ExitInstance()函数和线程运行 Run()函数。InitInstance()函数用于初始化 m_bAutoDelete 变量，使线程不自动退出，初始化 m_bDone 变量，启动 Run()函数的运行。ExitInstance()函数用于关闭串口。Run()函数使用 WaitCommEvent()函数监控串口事件，接收来自串口的数据，并发送消息给主对话框。

18.2.3　打开和关闭串口

打开串口由 CThreadCom 对象的 OpenCom()函数完成。它实现根据程序配置，打开指定工作串口的功能。代码如下。

```
01  BOOL CThreadCom::OpenCom(CString strCom, CWnd *pWndParent,
02      DWORD dwSendMsgToParent, DWORD dwRecvMsgToParent)//打开串口
03  {
04      CloseCom();                                     //先关闭串口
05      CString strLog;                                 //定义日志变量
06      m_hCom=::CreateFile(strCom,GENERIC_READ|GENERIC_WRITE,0, NULL,
07          OPEN_EXISTING,FILE_FLAG_OVERLAPPED, NULL); //打开串口
08      if (m_hCom==INVALID_HANDLE_VALUE)               //如果打开串口失败
09      {
10          strLog.Format("Open %s Error",strCom);      //格式化日志消息
11          AfxMessageBox(strLog);                      //弹出消息提示框
12          return false;                               //函数错误返回
13      }
14      ::SetupComm(m_hCom,MAXCOMINBUF,MAXCOMOUTBUF);   //设置串口输入输出缓
15                                                      //冲区
16      DCB dcb;                                        //定义 DCB 结构
17      if (!GetCommState(m_hCom,&dcb))                 //获取当前 DCB 结构的
18                                                      //值，如果失败
19      {
20          AfxMessageBox("获取串口状态错误!");           //显示错误提示消息框
21          CloseHandle(m_hCom);                        //关闭串口句柄
22          m_hCom=INVALID_HANDLE_VALUE;                //设置串口句柄无效
```

```
23            return false;                          //函数错误返回
24        }
25        if (!SetCommState(m_hCom,&dcb))            //设置串口参数，如果失败
26        {
27            CloseHandle(m_hCom);                   //关闭串口句柄
28            m_hCom=INVALID_HANDLE_VALUE;           //设置串口句柄无效
29            strLog.Format("Set %s CommState Error!",strCom);//格式化日志消息
30            AfxMessageBox(strLog);                 //函数错误返回
31            return false;
32        }
33        m_sError="No Error";                       //初始化错误消息变量
34        m_pWndParent    =   pWndParent;            //存储父窗口句柄
35        m_dwSendMsgToParent = dwSendMsgToParent;   //存储接收发送消息的父
36                                                   //窗口
37        m dwRecvMsgToParent = dwRecvMsgToParent;   //存储接收接收消息的父
38                                                   //窗口
39        DWORD CommMask;                            //定义串口事件标记
40        CommMask=0
41            |   EV_BREAK                           //检测到中断信号
42            |   EV_CTS                     //检测到 CTS 信号发生改变
43            |   EV DSR                     //检测到 DSR 信号发生改变
44            |   EV_ERR                     //发生行状态错误
45            |   EV_EVENT1                  //第一个提供程序指定的事件
46            |   EV_EVENT2                  //第二个提供程序指定的事件
47            |   EV_PERR                    //发生打印错误
48            |   EV_RING                    //检测到振铃
49            |   EV_RLSD                    //检测到 RLSD 信号发生改变
50            |   EV_RX80FULL                //接收缓冲区中数据达到 80%
51            |   EV RXCHAR                  //接收到字符，并放入输入缓冲区
52            |   EV_RXFLAG                  //接收到事件字符，并放入输入缓冲区
53            |   EV_TXEMPTY;                //输出缓冲区中的最后一个字符发送
54                                           //出去
55        ::SetCommMask(m_hCom,CommMask);    //注册要处理的事件
56        ::GetCommTimeouts(m hCom,&m Commtimeout);  //获取超时时间设置
57        m_Commtimeout.ReadTotalTimeoutMultiplier=5;//设置读操作超时时间
58        m_Commtimeout.ReadTotalTimeoutConstant=100;//设置读操作超时常量
59        m_bInit=true;                              //初始化串口状态为 true
60        return true;                               //函数成功返回
61    }
```

关闭串口由 CThreadCom 对象的 CloseCom()函数完成。它实现关闭指定工作串口的功能。代码如下：

```
01    BOOL CThreadCom::CloseCom()                //关闭串口函数
02    {
03        if (m_hCom!=INVALID_HANDLE_VALUE)      //如果串口句柄有效
04        {
05            PurgeComm(m_hCom,PURGE_RXCLEAR);   //释放串口事件
06            CloseHandle(m hCom);               //关闭串口句柄
07            m_hCom=INVALID_HANDLE_VALUE;       //设置串口句柄无效
08        }
09        m_bInit=false;                         //设置串口初始化状态为 false
10        return true;                           //函数成功返回
11    }
```

其中，CloseCom()函数调用 CloseHandle()函数关闭串口句柄。OpenCom()函数调用 CreateFile()函数打开串口，调用 SetupComm()函数设置串口输入输出缓冲区的大小，调用 SetCommState()函数设置串口资源的状态，最重要的是调用 SetCommMask()函数设置监控的事件。打开串口后，就可以向串口发送数据。

18.2.4　向串口发送数据

向串口发送数据由 CThreadCom 对象的 SendData()函数完成。它实现向指定串口发送指定长度数据的功能。代码如下：

```
01  BOOL CThreadCom::SendData(BYTE *s, DWORD dwLen)//发送数据函数
02  {
03      if (!dwLen) return true;                //如果数据长度为 0，则函数返回
04      ::GetCommTimeouts(m_hCom,&m_Commtimeout);       //获取超时时间设置
05      m_Commtimeout.WriteTotalTimeoutMultiplier=0;//设置读操作超时时间
06      m_Commtimeout.WriteTotalTimeoutConstant=2*dwLen;
07                                                      // 设 置 读 操 作 超 时
08                                                      //常量
09      ::SetCommTimeouts(m_hCom,&m_Commtimeout);   //设置超时时间设置
10      if (m_hCom!=INVALID_HANDLE_VALUE)           //如果打开串口失败
11      {
12          DWORD dwSend;                           //定义发送数据个数变量
13
14      m_pWndParent->SendMessage(m_dwSendMsgToParent,(DWORD)s,dwLen);
15                                                  //发送日志
16          if (!WriteFile(m hCom,s,dwLen,&dwSend,&m overWrite))
17                                                  //向串口发送数据
18          {
19              m_sError="串口发送数据错误";            //格式化错误消息
20              return false;                       //函数错误返回
21          }
22          return true;                            //发送完成后，函数成功返回
23      }
24      else                                        //串口句柄无效
25      {
26          m_sError="串口句柄无效";                  //格式化错误消息
27          return false;                           //函数错误返回
28      }
29  }
```

上面代码使用 SetCommTimeouts()函数设置写操作的超时时间，并调用 WriteFile()函数向串口发送数据。这样在程序中就可以调用此类实现串口操作。

18.2.5　界面处理

在程序界面上有“打开串口”的命令按钮，由 CCommSampleDlg 类的 OnButtonOpen()函数完成。它实现选择打开串口的功能。代码如下：

```
01  //打开串口
02  void CCommSampleDlg::OnButtonOpen()
03  {
```

```
04     if (pThreadCom != NULL)  return;
05     CString str;
06     CString com = "COM3";
07     //启动串口线程
08     pThreadCom = (CThreadCom*)
09                     AfxBeginThread(RUNTIME_CLASS(CThreadCom));
10     pThreadCom->SetComStr(com);     //设置串口线程的串口名称变量
11
12     if (pThreadCom->OpenCom(com, (CWnd*)this->GetSafeOwner(),
13         WM USER COMSENDMESSAGE, WM USER COMRECVMESSAGE))
14     {
15         str.Format("打开串口%s 成功", pThreadCom->GetComStr());
16         WriteLog(str);
17     }
18     else
19     {
20         str.Format(pThreadCom->m_sError+",请重新配置串口!");
21         WriteLog(str);
22         pThreadCom->m_bInit=FALSE;      //设置串口线程状态为 false
23         return ;
24     }
25     m_bCom=TRUE;
26     return ;
27 }
```

在程序界面上有“发送数据”的命令按钮，由 CCommSampleDlg 类的 OnButtonSend() 函数完成。它实现向指定串口发送指定数据的功能。代码如下：

```
01 void CCommSampleDlg::OnButtonSend()         //发送串口数据
02 {
03     UpdateData(true);                       //从控件更新数据变量
04     Int iLen = m_editSend.GetLength();      //获取发送文本框中的数据个数
05     BYTE* s= new BYTE[iLen];                //定义数据缓冲区
06     memset(s, 0x00, iLen);                  //初始化数据缓冲区
07     memcpy(s, (LPCTSTR)m_editSend, iLen);   //复制数据
08     //调用串口线程类向串口发送数据
09     pThreadCom->SendData((unsigned char*)s, iLen);
10 }
```

在串口发送数据和接收到数据后，界面可以通过这两个消息处理这两种情况。在本例中，由 OnSendMsg()函数处理发送数据的消息，OnRecvMsg()函数处理接收数据的消息。它们分别实现在界面上显示发送的数据和接收的数据的功能。代码如下：

```
01 //发送数据通知
02 void CCommSampleDlg::OnSendMsg(DWORD dwEvent,DWORD dwLen)
03 {
04     if(!dwLen)  return;                     //如果数据长度为 0，则函数返回
05     BYTE* temp = new BYTE[dwLen+1];         //定义数据缓冲区
06     memset(temp, 0x00, dwLen+1);            //初始化数据缓冲区
07     memcpy(temp, (const void*)dwEvent, dwLen);//复制数据
08     CString log;                            //定义日志变量
09     log.Format("\r\n 发送数据=%s", (LPCTSTR)temp);//格式化日志消息
10     if (m editLog)                          //如果存在日志文本框
11     {
12         CEdit* editLog=(CEdit*)FromHandle(m_editLog);
13                                             //获取日志文本框句柄
```

```
14          if (editLog->GetWindowTextLength()>50000)
15                                          //如果日志文本框内容超过 50000 字节
16
17          {
18              editLog->SetSel(0,-1);          //选择全部内容
19              editLog->Clear();               //清空全部内容
20              editLog->SetSel(0,0);           //设置位置为 0 字节处
21              editLog->ReplaceSel(log);       //使用日志消息替换原来的内容
22          }
23          else                                //如果日志文本框不满
24          {
25              editLog->SetSel(editLog->GetWindowTextLength(),
26                  editLog->GetWindowTextLength());//选中尾部
27              editLog->ReplaceSel(log);       //添加新内容
28          }
29      }
30      return;                                 //函数返回
31  }
32  //接收消息通知
33  void CCommSampleDlg::OnRecvMsg(DWORD dwEvent,DWORD dwLen)
34  {
35      if(!dwLen)  return;                     //如果数据长度为 0，则函数返回
36      BYTE* temp = new BYTE[dwLen+1];         //定义数据缓冲区
37      memset(temp, 0x00, dwLen+1);            //初始化数据缓冲区
38      memcpy(temp, (const void*)dwEvent, dwLen);//复制数据
39      CString log;                            //定义日志变量
40      log.Format("\r\n 接收数据=%s", (LPCTSTR)temp);//格式化日志消息
41      if (m_editRecv.GetLength() > 50000)     m_editRecv = "";
42                                              //如果接收文本框满了，则清空
43      m_editRecv += log;                      //将日志追加到接收文本框
44      UpdateData(false);                      //更新控件数据显示
45      return;                                 //函数返回
46  }
```

在上面的代码中，OnButtonOpen()函数是“打开串口”按钮的处理函数，它调用上面介绍的串口线程类的 OpenComm()函数打开 COM1。OnSendMsg()函数和 OnRecvMsg()函数分别是串口发送数据和接收数据的通知函数，此处的处理，是分别在日志文本框中和接收数据文本框中显示发送的数据和接收的数据。OnButtonSend()函数是“发送”按钮的处理函数，它调用串口线程类的 SendData()函数发送数据。程序运行效果如图 18-4 所示。



图 18-4　通信端口编程实例运行效果

18.3 本章小结

本章主要讲述了在 Windows 下进行串行端口和并行端口的编程方法。本章的重点是理解 Windows 下串行端口编程的流程和参数的设置，以及如何编写高效串行端口程序的方法。本章的难点是掌握串口数据流传输的控制机制。第 19 章将讲解有关 Internet 编程的方法。

18.4 习　题

依据以下两点比较 18.1.8 小节和 18.2 节中示例的异同：

（1）运行的方式，即两个示例的使用方式是一样的吗？

（2）串口编程的流程，即两个示例对串口的操作流程一致吗？

【思路】实际地运行、使用程序，然后找到串口编程的代码，总结串口编程的流程，并进行比较。

第 19 章　Internet 编程

网络互联是信息时代发展的趋势，由此产生了 Internet 编程。Internet 编程分为 Internet 客户端程序和 Internet 服务器程序。为了简化开发 Internet 程序的步骤，MFC 提供了 WinInet 技术支持 Internet 客户端编程，提供 ISAPI 扩展支持创建 Internet 服务器应用程序，并且 MFC 提供了 MAPI 编程技术，支持用户通过 E-mail 发送文档数据。本章将介绍这 3 个方面的知识。

19.1　WinInet 编程

通过 Internet 客户端应用程序可以访问万维网或 Intranet 局域网中的各种资源和数据。MFC 使用 WinInet 编程技术提供对编写使用 Internet 协议访问网络数据源的应用程序的支持。WinInet 中提供了对 Gopher 协议、FTP 协议和 HTTP 协议的支持。本节将介绍有关 WinInet 编程的方法。

19.1.1　WinInet API 概述

WinInet API 提供了抽象底层协议的高层接口，可以很容易地创建访问标准 Internet 协议的独立的应用程序。此程序也就是 Internet 客户端应用程序，可以通过 Internet 协议访问网络数据源的信息。如 Internet 客户端从服务器获取天气信息、新闻头条和商品价格等各种数据。

MFC 封装的这些 Internet 扩展使用一组标准的、易于使用的类实现。可以通过直接调用 Win32 函数或使用 MFC 的 WinInet 类编写 WinInet 客户端应用程序。使用 WinInet，编程人员不需要处理 WinSock、TCP/IP 或指定的网络协议的细节，简化了开发工作量。并且 WinInet 为 Gopher 协议、FTP 协议和 HTTP 协议提供统一的相近的 Win32 API 接口的函数集。因此，当底层协议改变时，代码修改量很少。WinInet API 函数如表 19-1 所示。

表 19-1　WinInet API函数

WinInet API 函数	功　　能
InternetAttemptConnect()	此函数用于在执行查询请求前第一次建立连接。客户端程序使用此函数可以调用建立拨号对话框。如果此函数失败，则应用程序进入脱机模式
InternetCheckConnection()	使用此函数检测是否可以建立到指定 URL 的连接
InternetCloseHandle()	关闭 Internet 句柄
InternetConfirmZoneCrossing()	检测原来的 URL 与新 URL 之间是否发生改变，如果两个 URL 之间发生安全性改变，应用程序会发送改变通知用户，通常是显示提示对话框

续表

WinInet API 函数	功　能
InternetConnect()	打开一个到 FTP、Gopher 或 HTTP 站点的会话
InternetErrorDlg()	显示错误对话框
InternetFindNextFile()	查找下一个 FTP 文件或 Gopher 文件
InternetGetLastResponseInfo()	获取最近一次的 Internet()函数发生的错误描述或服务器返回的响应信息
InternetLockRequestFile()	锁定使用的数据
InternetOpen()	初始化 Win32 的 Internet()函数
InternetQueryDataAvailable()	查询可用数据数目
InternetQueryOption()	查询指定 Internet 句柄的选项
InternetReadFile()	从使用 InternetOpenUrl()、FtpOpenFile()、GopherOpenFile()或 HttpOpenRequest()函数打开的句柄读取数据
InternetReadFileEx()	从使用 InternetOpenUrl()、FtpOpenFile()、GopherOpenFile()或 HttpOpenRequest()函数打开的句柄读取数据
InternetSetFilePointer()	设置 InternetReadFile()函数的文件位置
InternetSetOption()	设置 Internet 选项
InternetSetOptionEx()	设置 Internet 选项
InternetSetStatusCallback()	设置执行 Internet()函数执行操作时的回调函数
InternetTimeFromSystemTime()	按指定的 RFC 格式格式化日期和时间
InternetTimeToSystemTime()	获取 HTTP 返回的日期/时间字符串，并将其转换为 SYSTEMTIME 结构
InternetUnlockRequestFile()	解锁使用的文件
InternetWriteFile()	写数据到打开的 Internet 文件

使用上面列出的 WinInet API 函数可以实现对使用 HTTP 协议、FTP 协议和 Gopher 协议服务器的数据访问。

19.1.2　WinInet 常用类概览

19.1.1 小节介绍了 WinInet API 函数，使用这些函数编写的程序有很多地方是相似的，为此，MFC 封装了如表 19-2 所示的类和全局函数以支持 Internet 客户端应用程序的编写，以提高开发效率。

表 19-2　WinInet类

类　名	功　能
CInternetSession	此类用于创建和初始化单个或几个同步的 Internet 会话，还提供连接到代理服务器的信息。如果创建的 Internet 连接在整个应用程序中有效，则可以在应用程序类中创建此变量的对象。可以调用此类的 OpenURL()函数打开 URL。SetCookie()成员函数、GetCookie()成员函数和 GetCookieLength()成员函数提供管理 cookie 的功能。此类是编写 WinInet 程序的第一步
CInternetConnection	此类用于管理到 Internet 服务器的连接，是 CFtpConnection 类、CHttpConnection 类和 CGopherConnection 类的基类。这 3 个类在此类的基础上分别提供处理 FTP、HTTP 和 GOPHER 服务器的附加功能的函数。要直接与 Internet 服务器进行通信，则必须有一个 CInternetSession 对象和一个 CInternetConnection 对象
CFtpConnection	此类既提供管理与 FTP 网络服务器相连的连接，又提供直接操作服务器上的目录和文件的功能。要与 FTP 网络服务器进行通信，则首先需要创建一个 Cinternet-Session 实例，然后，创建一个 CFtpConnection 对象。当创建 CFtpConnection 对象时，必须使用 CInternetSession 类的 GetFtpConnection()函数，创建一个 CFtpConnection 对象，并返回创建的指针

续表

类　　名	功　　能
CGopherConnection	此类用于管理与 gopher 网络服务器相连的连接。要与 gopher 网络服务器进行通信，则首先需要创建一个 CInternetSession 实例，然后，创建一个 CGopher-Connection 对象。在创建 CGopherConnection 对象时，必须使用 CInternetSession 类的 GetGopherConnection()函数，创建一个 CGopherConnection 对象，并返回创建的指针。此类包含构造函数和 3 个成员函数管理 Gopher 服务，OpenFile()函数用于打开文件，CreateLocator()函数用于创建定位器，GetAttribute()函数用于获取属性
CHttpConnection	此类用于管理与 HTTP 网络服务器相连的连接。要与 HTTP 网络服务器进行通信，则首先需要创建一个 CInternetSession 实例，然后，创建一个 CHttpConnection 对象。在创建 CHttpConnection 对象时，必须使用 CInternetSession 类的 GetHttp-Connection()函数，创建一个 CHttpConnection 对象，并返回创建的指针。此类的 OpenRequest()函数用于管理使用 HTTP 协议的到服务器的连接
CInternetFile	此类提供 CHttpFile 类和 CGopherFile 类的基类。这两个类允许使用网络协议访问远程计算机上的文件。当创建 CInternetFile 对象时，必须通过 CGopher-Connection 对象的 OpenFile()函数、CHttpConnection 对象的 OpenRequest()函数或 CFtpConnection 对象的 OpenFile()函数创建。此类具有 Open()函数、LockRange()函数、UnlockRange()函数和 Duplicate()函数，这 4 个函数是虚函数，继承类需要重载这几个函数
CGopherFile	此类提供在 Gopher 服务器上查找和读取文件的功能。Gopher 服务器不支持向 Gopher 文件写入数据，因为此服务函数主要使用菜单接口查找信息。因此，此类不支持 Write()函数、WriteString()函数和 Flush()函数
CHttpFile	此类提供在 HTTP 服务器上请求和读取文件的功能。如果程序中的 Internet 会话要从 HTTP 服务器上读取数据，则必须创建 CHttpFile 实例
CFileFind	此类完成本地文件的查找，是 CGopherFileFind 类和 CFtpFileFind 类的基类，这两个类是完成 Internet 文件查找功能的类。此类包含开始查询、定位文件、返回标题和返回文件名等函数。对于 Internet 文件查找，GetFileURL()成员函数返回文件的 URL。没有实现在 HTTP 服务器上进行查找的类，因为 HTTP 服务器不支持文件查找
CFtpFileFind	此类实现在 FTP 服务器上进行 Internet 文件的查找。包括开始查找、定位文件和获取 URL 或有关文件的其他描述信息的函数
CGopherFileFind	此类实现在 Gopher 服务器上进行 Internet 文件的查找。包括开始查找、定位文件和获取文件 URL 的函数
CGopherLocator	此类从 Gopher 服务器上获取 Gopher“定位符”，确定定位符类型，并构造可以用在 CGopherFileFind 类中的定位符格式。应用程序在从 Gopher 服务器上接收信息前，必须首先获取 Gopher 服务器的定位符。Gopher 定位符中有属性可以确定查找到的服务器或文件的类型。通常应用程序在 CGopherFileFind::FindFile 文件中使用定位符获取指定类型的信息
CInternetException	此对象表示与 Internet 操作相连的异常情况。此类包含两个成员，一个是与异常相关的错误代码，一个是存放与错误相连的 Internet 应用程序的上下文

除了提供了上面的这些 WinInet 类外，MFC 还提供了下面 3 个全局函数支持 WinInet 的编程。

（1）AfxParseURL()函数用在 CInternetSession::OpenURL()函数中，解析 URL 字符串并返回服务类型和相应的组件。其函数原型为：

```
BOOL AFXAPI AfxParseURL(
  LPCTSTR pstrURL,                    //指向包含要解析的 URL 的字符串的指针
  DWORD& dwServiceType,               //指定要解析的格式
```

```
  CString& strServer,                    //存放 URL 服务类型后的第一部分
  CString& strObject,                    //存放 URL 指向的对象
  INTERNET_PORT& nPort );                //存放服务器或对象的端口
```

其中，dwServiceType 参数的有效值如表 19-3 所示。

表 19-3　协议解析格式

解析格式值	含　义
AFX_INET_SERVICE_FTP	FTP 协议
AFX_INET_SERVICE_HTTP	HTTP 协议
AFX_INET_SERVICE_HTTPS	HTTPS 协议
AFX_INET_SERVICE_GOPHER	Gopher 协议
AFX_INET_SERVICE_FILE	文件协议
AFX_INET_SERVICE_MAILTO	MAILTO 协议
AFX_INET_SERVICE_NEWS	NEWS 协议
AFX_INET_SERVICE_NNTP	NNTP 协议
AFX_INET_SERVICE_TELNET	TELNET 协议
AFX_INET_SERVICE_WAIS	WAIS 协议
AFX_INET_SERVICE_MID	MID 协议
AFX_INET_SERVICE_CID	CID 协议
AFX_INET_SERVICE_PROSPERO	PROSPERO 协议
AFX_INET_SERVICE_AFS	AFS 协议
AFX_INET_SERVICE_UNK	默认协议

如果传入的 URL 是有效格式，并且解析成功，则返回值为非 0，否则返回 0。如下所示的 URL：

```
http://127.0.0.1/root/index:8080
```

使用此函数解析后结果为：

```
strServer        == "127.0.0.1"
strObject        == "/root/index "
nPort            == 8080
dwServiceType    == http
```

（2）AfxGetInternetHandleType()函数可以根据传入的 Internet 句柄获取 Internet 句柄的类型。其函数原型如下：

```
DWORD AFXAPI AfxGetInternetHandleType(
  HINTERNET hQuery );                          //指定要查询句柄类型的句柄
```

函数的返回值表示输入的 Internet 句柄的句柄类型，其有效取值如表 19-4 所示。

表 19-4　WinInet句柄类型

句 柄 类 型	含　义
INTERNET_HANDLE_TYPE_INTERNET	Internet 会话句柄
INTERNET_HANDLE_TYPE_CONNECT_FTP	FTP 连接句柄
INTERNET_HANDLE_TYPE_CONNECT_GOPHER	Gopher 连接句柄
INTERNET_HANDLE_TYPE_CONNECT_HTTP	HTTP 连接句柄
INTERNET_HANDLE_TYPE_FTP_FIND	FTP 查找句柄

续表

句 柄 类 型	含　　义
INTERNET_HANDLE_TYPE_FTP_FIND_HTML	FTP 的 HTML 查找句柄
INTERNET_HANDLE_TYPE_FTP_FILE	FTP 文件句柄
INTERNET_HANDLE_TYPE_FTP_FILE_HTML	FTP 的 HTML 文件句柄
INTERNET_HANDLE_TYPE_GOPHER_FIND	Gopher 查找句柄
INTERNET_HANDLE_TYPE_GOPHER_FIND_HTML	Gopher 的 HTML 查找句柄
INTERNET_HANDLE_TYPE_GOPHER_FILE	Gopher 文件句柄
INTERNET_HANDLE_TYPE_GOPHER_FILE_HTML	Gopher 的 HTML 文件句柄
INTERNET_HANDLE_TYPE_HTTP_REQUEST	HTTP 请求句柄

如果传入的句柄为 NULL 或是无效类型，则函数返回 AFX_INET_SERVICE_UNK。

（3）AfxThrowInternetException()函数用于抛出 Internet 操作异常，抛出的错误代码应该与 Windows 操作系统的错误代码一致。其函数原型为：

```
void AFXAPI AfxThrowInternetException(
  DWORD dwContext,                    //指定发生错误的操作的上下文
  DWORD dwError = 0 );                //指定发生的错误的错误代码
```

MFC 通过这些类和这 3 个全局函数提供对 WinInet 编程的知识。从 19.1.3 小节开始，将分别讲述 HTTP、FTP 和 Gopher 的编程。

19.1.3　超文本传输协议 HTTP 编程

HTTP（Hypertext Transfer Protocol）协议，即超文本传输协议，是目前互联网上最常用的通信协议，也是 TCP/IP 协议层上的传输协议。本小节结合 19.1.2 小节介绍的 WinInet 类，以一个示例介绍 HTTP 编程。在本小节的示例中，会下载用户指定的 HTTP 页面。代码如下。

```
01  void CHTTPSampleView::OnButtonDownload()        //下载 HTTP 页面
02  {
03      UpdateData(true);                           //从数据控件中获取数据
04      CString strServerName;                      //服务器名称
05      CString strObject;                          //地址对象
06      INTERNET_PORT nPort;                        //端口
07      DWORD dwServiceType;                        //协议类型
08      if (!AfxParseURL(m_Address, dwServiceType, strServerName,
09                          strObject,  nPort))     //解析 URL
10      {   //如果失败，则可能是没有加 http://
11          m_Address = _T("http://") + m_Address; //在地址前加入 http 头
12          //解析 URL
13          if (!AfxParseURL(m_Address, dwServiceType, strServerName,
14                              strObject, nPort))
15          {
16              AfxMessageBox("无效的 URL", MB_OK); //显示错误消息框
17              return;                             //函数操作成功，返回
18          }
19      }
20      CWaitCursor cursor;                         //显示等待光标
21      CInternetSession session("HTTP Session");  //定义 HTTP 会话变量
```

```
22      if (m_pHttpConnection != NULL)
23          m_pHttpConnection->Close();               //如果 HTTP 连接有效，则关闭
24      delete m_pHttpConnection;                     //删除 HTTP 连接
25      m_pHttpConnection = NULL;                     //赋值 HTTP 连接为 NULL
26      CHttpFile* pFile = NULL;                      //定义 CHttpFile 变量
27      CString strHeader;                            //定义信息头变量
28      DWORD dwReturn;                               //定义返回值变量
29      try
30      {   //从 HTTP 会话中，获取 HTTP 连接
31          m_pHttpConnection = session.GetHttpConnection(strServerName,
32                                  nPort);
33          //打开 HTTP 连接，并获取文件对象
34          pFile = m_pHttpConnection->OpenRequest(
35              CHttpConnection::HTTP_VERB_GET, strObject);
36          pFile->AddRequestHeaders(strHeader);      //增加信息头
37          pFile->SendRequest();                     //发送请求
38          pFile->QueryInfoStatusCode(dwReturn);     //查询返回的状态值
39          if (dwReturn == HTTP_STATUS_OK)           //如果请求返回成功
40          {
41              char szBuff[1024];                    //定义数据数组
42              UINT nRead = pFile->Read(szBuff, 1023);//从文件中读取数据
43              //循环处理，依次读取文件内容
44              while (nRead > 0)
45              {
46                  m_Content += szBuff;//将读取的数据保存到 m_Content 变量中
47                  nRead = pFile->Read(szBuff, 1023); //继续读取文件内容
48              }
49          }
50          delete pFile;                             //删除文件对象
51          delete m_pHttpConnection;                 //删除 HTTP 连接
52          AfxMessageBox("下载 HTTP 页面成功");      //显示消息提示框
53      }
54      catch (CInternetException* pEx)               //从 WinINet 中检查错误
55      {
56          TCHAR szErr[1024];                        //定义消息变量
57          if (pEx->GetErrorMessage(szErr, 1024))    //获取错误信息
58              AfxMessageBox(szErr, MB_OK);          //显示信息提示框
59          else
60              AfxMessageBox("下载 HTTP 页面失败", MB_OK);//显示错误提示
61          pEx->Delete();                        //删除异常变量
62          m_pHttpConnection = NULL;             //设置 HTTP 连接对象为 NULL
63      }
64      session.Close();                          //关闭会话
65      UpdateData(false);                        //将获取的文件内容，显示在控件中
66  }
```

上面代码中，从控件中获取用户输入的地址，并调用 AfxParseURL()函数解析输入的URL 地址是否是有效的。如果不是有效地址，则假定没有加协议前缀，在此基础上增加 http前缀，再进行解析。解析成功后，创建 CInternetSession 会话对象，并调用 GetHttpConnection()函数传入解析后的各项参数，如服务器名称、端口等信息，创建 CHttpConnection 对象。

接着调用连接对象的 OpenRequest()函数，打开 HTTP 请求，并保存返回的 CHttpFile 文件，调用文件对象的 Read()函数读取 HTTP 文件中的内容。最后关闭会话对象。程序运行效果如图 19-1 所示。

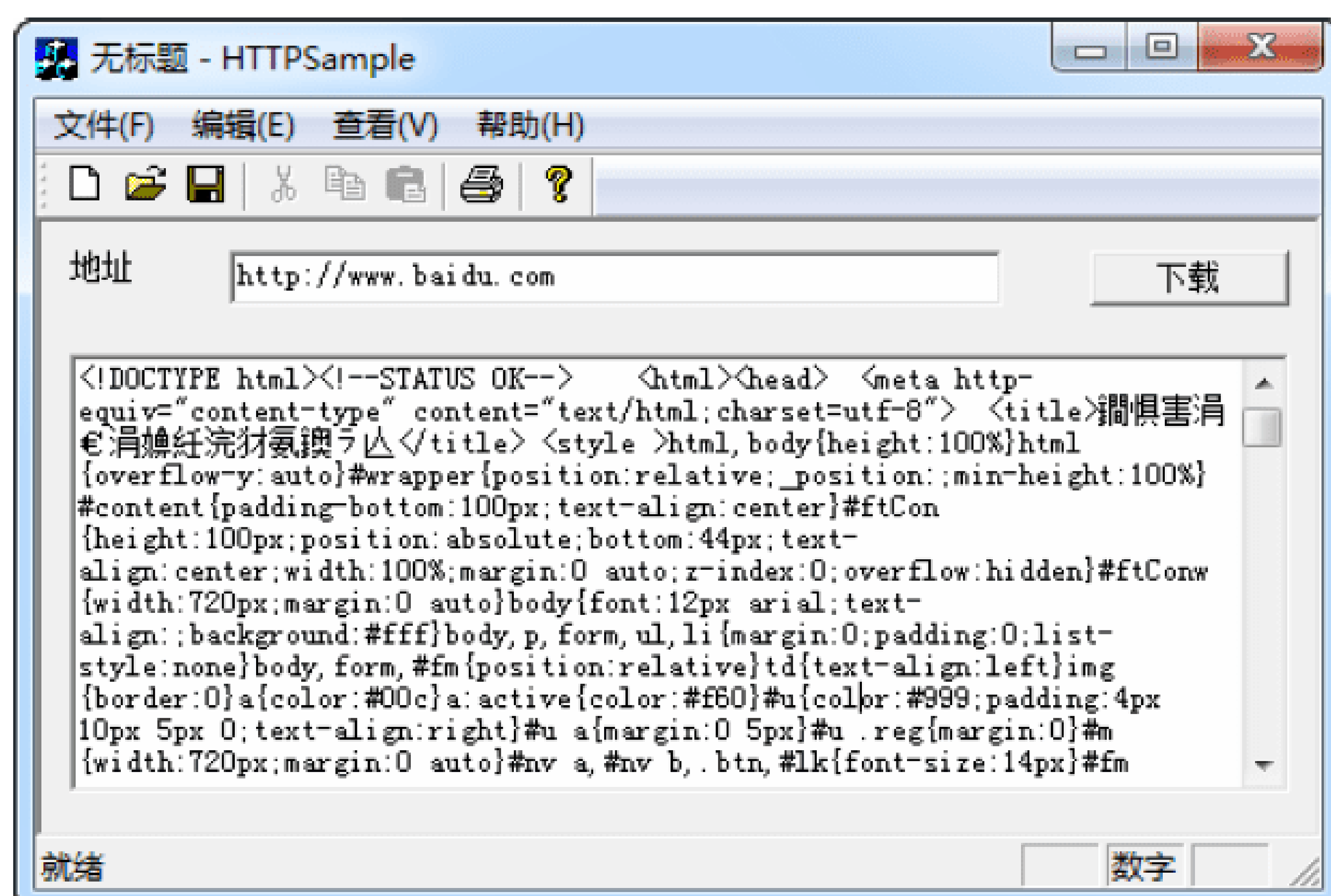


图 19-1　下载 HTTP 页面运行效果

19.1.4　文件传输协议 FTP 编程

FTP（File Transfer Protocol）协议，即文件传输协议，用于控制文件的双向传输。本小节结合 19.1.2 小节介绍的 WinInet 类，以一个示例介绍 FTP 编程。在本小节的示例中，会显示出用户指定的 FTP 站点的文件。代码如下：

```
01  void CFTPSampleView::OnButtonGoto()          //导航 FTP 站点
02  {
03      m_fileCtrl.ResetContent();               //清空文件列表控件中的数据项
04      CString strServerName;                   //定义服务器名称变量
05      CString strObject;                       //定义对象变量
06      INTERNET_PORT nPort;                     //定义端口变量
07      DWORD dwServiceType;                     //定义服务类型变量
08      UpdateData(true);                        //从控件中获取数据
09      CInternetSession session("FTP Session");    //定义 Internet 会话变量
10      if (m_pFtpConnection != NULL)   m_pFtpConnection->Close();
11                                               //如果 FTP 连接有效，则关闭
12      delete m_pFtpConnection;                 //删除 FTP 连接
13      m_pFtpConnection = NULL;                 //赋值 FTP 连接为 NULL
14      if (!AfxParseURL(m_Address, dwServiceType, strServerName,
15               strObject, nPort))              //解析 URL
16      {   //如果失败，则可能是没有加 ftp://
17          CString strFtpURL = _T("ftp://");    //定义 FTP 头字符串
18          m_Address = strFtpURL + m_Address; //在原有地址上增加FTP头字符串
19          if (!AfxParseURL(m_Address, dwServiceType, strServerName,
20                          strObject, nPort))
21          {   //解析 URL
22              AfxMessageBox(IDS_INVALID_URL, MB_OK); //提示错误消息
23              return;                          //函数返回
24          }
25      }
```

```
26      CWaitCursor cursor;                      //显示等待光标
27      if ((dwServiceType == INTERNET SERVICE FTP) &&
28              !strServerName.Is   Empty())
29      {   //如果服务类型是 FTP，并且服务器名称不为空，则打开 FTP 连接
30          try
31          {
32              m pFtpConnection = session.GetFtpConnection(
33                      strServerName,m UserName,
34                  m Password, nPort);         //打开 FTP 连接
35          }
36          catch (CInternetException* pEx)     //从 WinINet 中检查错误
37          {
38              TCHAR szErr[1024];                     //定义消息变量
39              if (pEx->GetErrorMessage(szErr, 1024)) //获取错误信息
40                  AfxMessageBox(szErr, MB_OK);       //显示信息提示框
41              else
42                  AfxMessageBox(IDS_EXCEPTION, MB_OK);//显示错误提示
43              pEx->Delete();                  //删除异常变量
44              m  pFtpConnection = NULL;       //设置 FTP 连接对象为 NULL
45          }
46      }
47      else
48          AfxMessageBox(IDS_INVALID_URL, MB_OK);
49                                              //提示无效的 URL 错误信息
50      if (m_pFtpConnection != NULL)
51          ShowFiles();                    //如果 FTP 连接有效，则显示文件
52  }
```

上面代码从控件中获取用户输入的地址，并调用 AfxParseURL()函数解析输入的 URL 地址是否是有效的。如果不是有效地址，则假定没有加协议前缀，在此基础上增加 ftp 前缀，再进行解析。解析成功后，创建 CInternetSession 会话对象，调用 GetFtpConnection()函数传入解析后的各项参数，如服务器名称、用户名、密码和端口等信息，创建 CFtp-Connection 对象。如果创建成功，则调用 ShowFiles()函数显示其中的文件列表，其代码如下：

```
01  void CFTPSampleView::ShowFiles()                   //显示 FTP 站点上的内容
02  {
03      CFtpFileFind ftpFind(m_pFtpConnection);      //定义 FTP 文件查找对象
04      CString strFileName;                         //定义文件名变量
05      BOOL bContinue = ftpFind.FindFile(_T("/*"));//查找根目录下的文件
06      while (bContinue)                            //循环查找
07      {
08          bContinue = ftpFind.FindNextFile();      //查找下一个文件
09          strFileName = ftpFind.GetFileName();     //获取文件名
10          m_fileCtrl.AddString(strFileName);       //在文件列表中添加文件名
11      }
12      ftpFind.Close();                             //关闭 FTP 文件查找对象
13  }
```

上面代码会显示当前 FTP 站点的目录下的所有文件。使用上个函数创建成功的 m_pFtp Connection 对象初始化 FTP 文件查找对象 CFtpFileFind，调用 FindFile()开始查找文件，在 while 循环中调用 FindNextFile()函数循环查找文件信息，调用 GetFileName()函数获取文件名称，并将获取的文件名称显示在列表控件中。程序运行效果如图 19-2 所示。

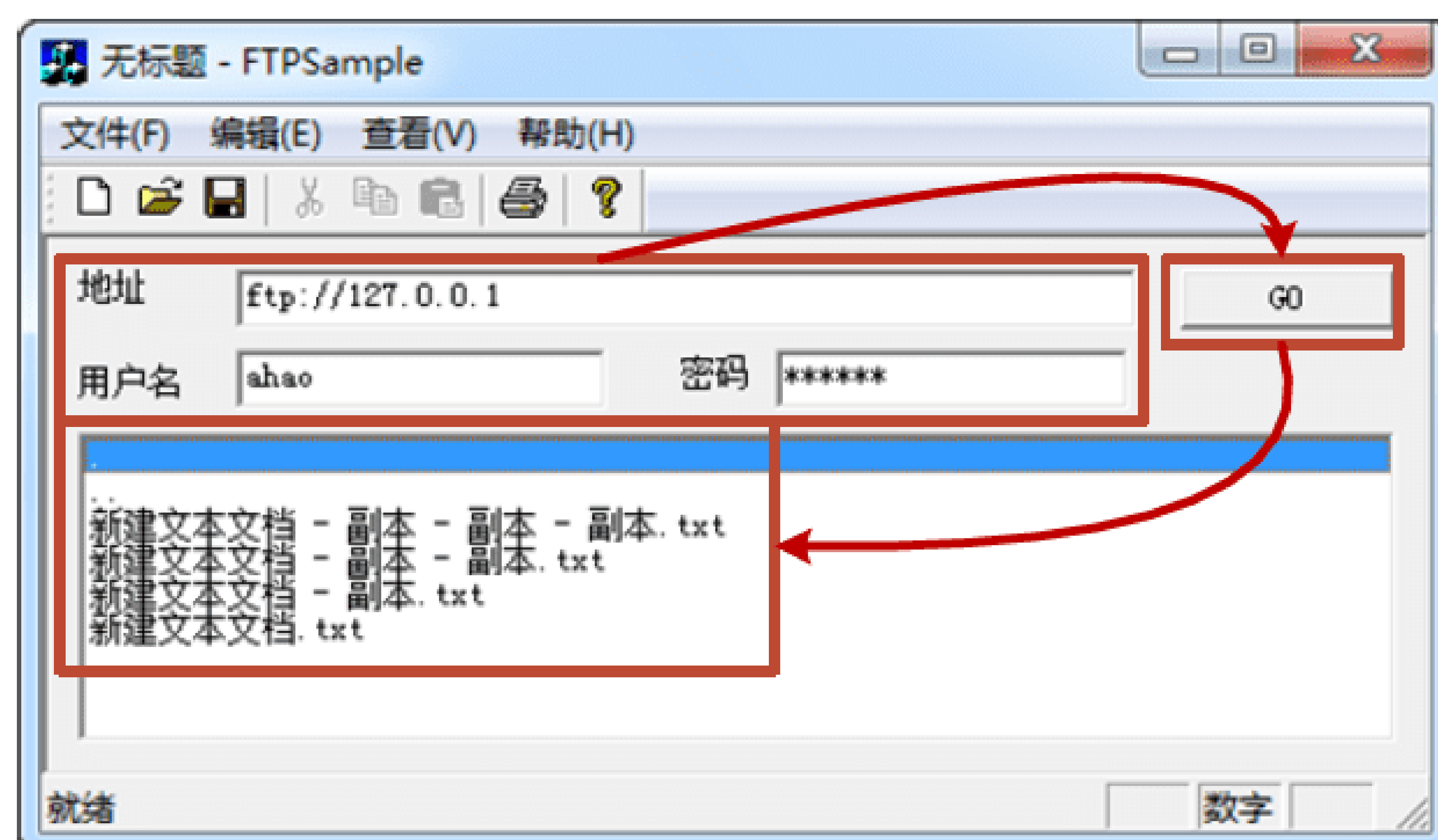


图 19-2　FTP 示例运行效果

19.1.5　网际 Gopher 协议编程

Gopher（Internet Gopher Protocol）协议，即网际 Gopher 协议，是目前互联网上非常有名的信息查询系统。使用菜单形式，可以实现信息检索。但是因为其局限性，现在 Gopher 协议已经比较少了。本小节结合 19.1.2 小节介绍的 WinInet 类，简单介绍 Gopher 编程。代码如下：

```
01  //连接 Gopher 站点
02  void CGopherSampleView::ConnectGopher(CString m_host)
03  {
04      UpdateData(true);
05      m_Content = "=========================================\r\n";
06      CWaitCursor cursor;                                //显示等待光标
07      CInternetSession session;
08      CGopherConnection* pConn = NULL;                   //定义连接对象
09      CGopherFileFind* pFile = NULL;
10      CString fileName;
11      try
12      {
13          pConn = session.GetGopherConnection(m_host);   //连接指定地址
14      }
15      catch (CInternetException* pEx)                    //捕获异常
16      {
17          pEx->ReportError();
18          pConn = NULL;
19          pEx->Delete();
20      }
21      if (pConn)                                      //判断连接是否成功
22      {
23          m_Content += "已建立链接。 \r\n";              //输入提示信息
24          CString line;
25          CGopherLocator locator = pConn->CreateLocator(NULL,
26          NULL, GOPHER_TYPE_DIRECTORY);                  //获取目录信息
27          line = locator;
28          //输出目录信息
29          m_Content += "第一个 Gopher 位置是" + line + "\r\n";
```

```
30          pConn->Close();                                   //关闭连接
31          delete pConn;                                     //删除连接
32      }
33      else
34      {
35          m_Content += "本地址没有发现 gopher 主机 。 \r\n"; //输入提示信息
36      }
37      m Content += "============================================\r\n";
38      UpdateData(false);                                    //更新控件显示
39  }
```

上面代码创建 CInternetSession 会话对象，并调用 GetGopherConnection()函数传入各项参数，如服务器名称、端口等信息。创建 CGopherConnection 对象，使用此对象初始化 CGopherLocator 对象，调用 CreateLocator()函数定位目录位置，并输出第一个目录信息，最后释放连接对象和文件查找对象并关闭会话对象，并将处理结果显示在控件中。程序运行效果如图 19-3 所示。

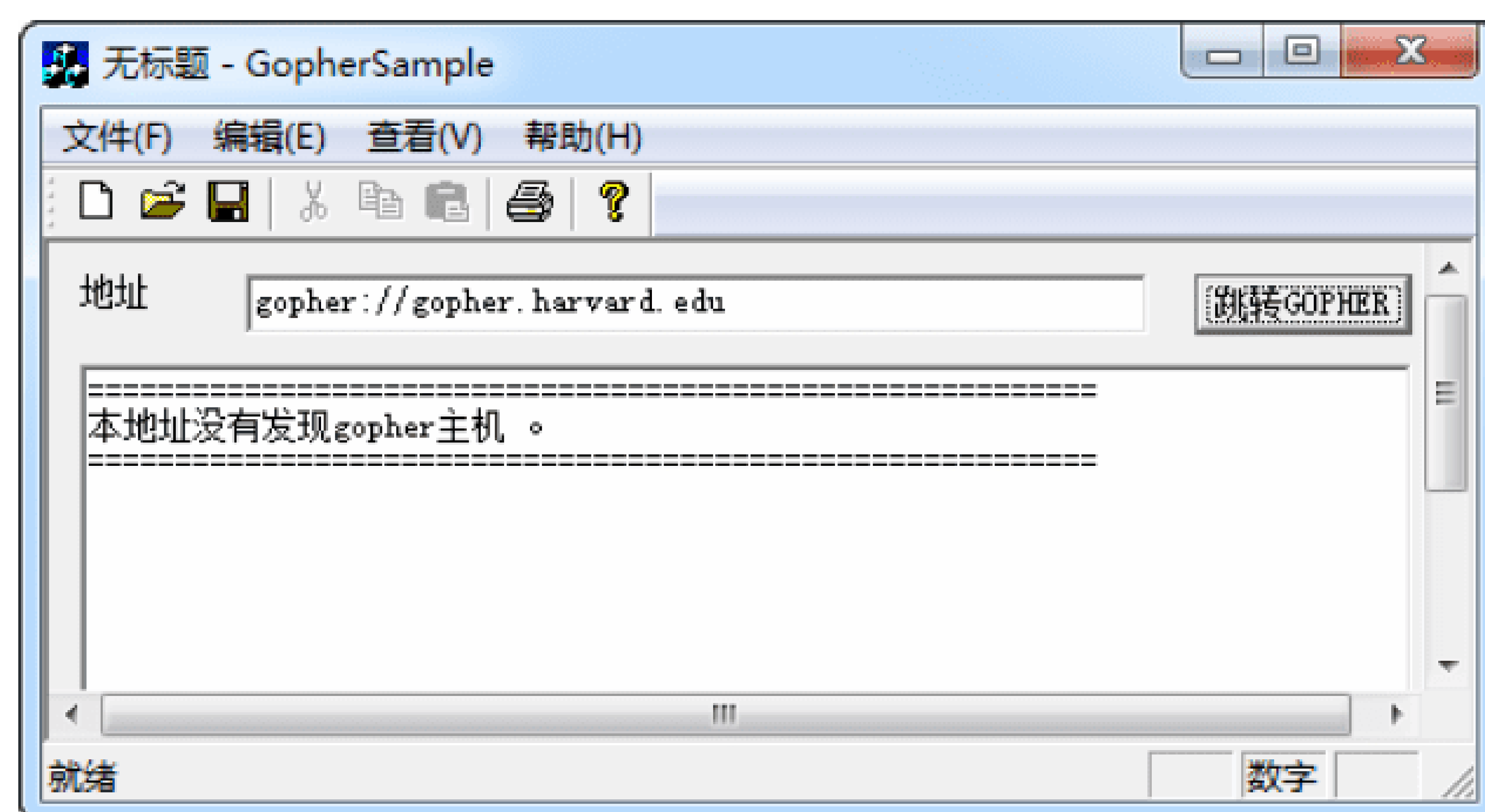


图 19-3　Gopher 浏览器运行效果

19.2　ISAPI 编程

使用 ISAPI 可以快速地开发 Internet 服务器应用程序，完成服务器对程序的控制。VC 为编写 ISAPI 应用提供了一组封装好的 MFC 类，并且提供了调试 ISA 和 ISAPI 程序的方法。本节将介绍 ISAPI 编程方法。

19.2.1　ISAPI 概述

ISAPI（Internet Server Application Programming Interface）即 Internet 服务器应用编程接口，是用于编写类似 IIS 等 Web 服务器的扩展 OLE 服务和过滤程序的一组 API 接口。要在服务器上运行 ISAPI，则 Internet 服务器必须支持超文本传输协议 HTTP 协议。使用具有 ISAPI 功能的 Web 服务器（如 Microsoft IIS），可以快速地创建 Internet 服务器应用程序。

MFC 封装了 ISAPI 的实现，使用 MFC 类可以快速地开发 ISAPI 扩展程序和过滤程序，并且只要 Internet 服务器软件具有 ISAPI 功能，就可以使用 MFC 创建的 Internet 服务器扩

展程序和过滤器程序。MFC 提供的 ISAPI 封装类如下。

- CHttpServer 类用于创建服务器扩展 DLL，也就是 Internet 服务器应用程序（ISA）。每个客户端命令对应 CHttpServer 对象的一个成员函数，并使用 EXTENSION_CONTROL_BLOCK 结构与服务器进行通信。每个 DLL 中只能包含一个 CHttpServer 对象，当有客户端发送请求时，创建新的 CHttpServerContext 对象与其进行通信。
- CHttpServerContext 类由 CHttpServer 对象创建，处理单个客户端到服务器的请求。因为服务器的处理必须是并发的，因此一个 CHttpServer 实例可以包含与之相连的多个 CHttpServerContext 对象。
- CHtmlStream 类对象由 CHttpServer 对象创建，用于管理对应的客户端缓冲区。CHtmlStream 可以使用 HTML 的开始标记、结束标记和内容标记格式化响应数据。
- CHttpFilter 类对象用于管理输入和输出客户端的数据通知过滤。一个 DLL 中，只能有一个 CHttpFilter 对象。当构造 CHttpFilter 对象时，会创建一个 CHttpFilterContext 对象管理与单个客户端相连的通知。因为服务器的处理是并发的，因此一个 CHttpFilter 实例可以与多个 CHttpFilterContext 对象相连。
- CHttpFilterContext 类对象由 CHttpFilter 对象创建，用于处理与单个客户端相连的过滤处理。

ISAPI 的应用包括 ISAPI 扩展服务程序和 ISAPI 过滤程序，这两者都是使用 ISAPI 编写，具有如下共同点。

- 这两种程序都共享服务的进程空间。
- 这两种程序都必须是线程安全的，因为 Internet 服务是多线程并发的。
- 这两种程序装载后，都始终存放在内存中，直到服务停止。

虽然两者有上面的共同点，但是，这两者又有如下区别。

- 调用时间不同：当引用 URL 时，ISAPI 服务器扩展程序才会被装载；而 ISAPI 过滤程序是服务器处理任何一个 URL 时都会调用。
- 调用方式不同：ISAPI 服务器扩展程序需要显示调用，如 http://127.0.0.1/my.dll?.;而只要是 ISAPI 过滤程序注册的事件，任何一个发生此事件的 URL 调用，都会自动执行。
- 装载时间不同：ISAPI 服务器扩展程序在用户第一次调用时装载；而 ISAPI 过滤程序是在服务启动时装载。

下面 3 个小节将分别介绍开发 ISAPI 服务器扩展程序、ISAPI 过滤程序以及 ISA 的调试。

19.2.2　ISAPI 服务器扩展程序

ISAPI 服务器扩展程序，也称为 Internet 服务器应用程序（ISA），是可以被 HTTP 服务器装载和调用的 DLL，可以增强 ISAPI 的功能和兼容性。ISA 由浏览器程序执行，可以将其理解为 CGI 程序。

使用 ISAPI 服务器扩展程序可以实现多种功能。如读者可以使用 ISAPI 编写读写数据库的订单系统。使用此种方式的优点是，用户不需要在每台客户端上安装应用程序。这样，程序更新非常容易。要运行一个 ISAPI 服务器扩展 DLL，用户在浏览器的地址栏上输入如

下形式的地址即可：

```
http://127.0.0.1/XXX.dll?
```

DLL 运行在服务器上，并发送 HTML 数据给客户端。使用这种方式，更新新版本 DLL，只需要停止服务，使用新版本的 DLL 替换旧版本的 DLL，并重启服务即可。当下一个客户端发送请求时，则会装载最新版本的 DLL 程序。这样要更新程序，只需要在服务器上更新一次即可，而不需要在客户端上多次安装。

使用 MFC 编写 ISA，第一步需要创建 CHttpServer 对象，每次接收到客户端请求时，创建一个 CHttpServerContext 对象，并将其指针传入命令处理函数中。使用多个 CHttpServerContet 对象可以实现多个客户端的并发。服务器扩展 DLL 包含以下两个导出函数。

- ❑ CHttpServer::GetExtensionVersion()函数：当服务器程序第一次运行时，会调用此函数，完成版本信息更新，并提供 ISA 程序的文本描述。
- ❑ CHttpServer::HttpExtensionProc()函数：类似于应用程序的 main()函数，可以通过回调函数读取客户端数据，并确定如何处理这些数据。当此函数返回时，服务器会格式化返回值并返回给客户端。

HTTP 服务器和 ISA 之间通过 EXTENSION_CONTROL_BLOCK 结构进行数据通信。此结构可以传输包括原查询字符串、路径信息、方法名称和解析路径等数据，传递给 CHttpServerContext 对象的 EXTENSION_CONTROL_BLOCK 结构定义如下：

```
typedef struct _EXTENSION_CONTROL_BLOCK {
    DWORD     cbSize;              //输入成员，表示此结构的大小
    DWORD     dwVersion            //输入成员，HTTP_FILTER_REVISION 的版本信息，
                                   //高位是主版本
    HCONN     ConnID;              //输入成员，HTTP 服务器分配的唯一编号，不可修改
    DWORD     dwHttpStatusCode;    //输出成员，当请求完成时，事务的状态
    CHAR      lpszLogData[HSE_LOG_BUFFER_LEN];     //输出成员，日志数据
    LPSTR     lpszMethod;                  //输入成员，请求的方法
    LPSTR     lpszQueryString;             //输入成员，请求字符串
    LPSTR     lpszPathInfo;                //输入成员，路径信息
    LPSTR     lpszPathTranslated;          //输入成员，转换后的路径
    DWORD     cbTotalBytes;                //输入成员，数据字节数
    DWORD     cbAvailable;                 //输入成员，有效字节数
    LPBYTE    lpbData;                     //输入成员，数据缓冲区
    LPSTR     lpszContentType;             //输入成员，内容类型
    BOOL  ( WINAPI * GetServerVariable ) //获取服务器变量的回调函数
       ( HCONN       hConn,                //连接句柄
        LPSTR       lpszVariableName,      //获取的变量名称
        LPVOID      lpvBuffer,             //存放变量值的缓冲区
        LPDWORD     lpdwSize );            //存放变量值的缓冲区的大小
    BOOL  ( WINAPI * WriteClient )         //向客户端写入数据
       ( HCONN      ConnID,                //连接句柄
       LPVOID     Buffer,                  //要写入的数据
       LPDWORD    lpdwBytes,               //要写入数据的长度
       DWORD      dwReserved );            //预留
    BOOL  ( WINAPI * ReadClient )          //从客户端读取数据
       ( HCONN      ConnID,                //连接句柄
       LPVOID     lpvBuffer,               //存放要读入数据的缓冲区
       LPDWORD    lpdwSize );              //要读入数据的长度
```

```
    BOOL ( WINAPI * ServerSupportFunction ) //服务器支持的函数
        ( HCONN     hConn,                  //连接句柄
        DWORD     dwHSERRequest,            //HTTP 服务器扩展值
        LPVOID    lpvBuffer,                //状态值缓冲区
        LPDWORD   lpdwSize,                 //状态值缓冲区大小
        LPDWORD   lpdwDataType );           //数据类型
} EXTENSION_CONTROL_BLOCK, *LPEXTENSION_CONTROL_BLOCK;
```

当服务器装载 DLL 时，会调用 DLL 的 CHttpServer::GetExtensionVersion()入口函数获取 HTTP_FILTER_REVISION 版本号和基本描述。每当有客户端连接，调用 CHttp Server::HttpExtensionProc()入口函数，并可以通过上面这个结构获取常用的信息，如查询字符串、地址信息以及方法名称等。

19.2.3　使用应用向导开发 ISAPI 服务器扩展程序

除了 19.2.2 小节介绍的 MFC 类外，VC 还提供了 ISAPI 扩展向导帮助创建 ISAPI 服务器扩展程序和 ISAPI 过滤程序。读者可以选择如何链接到 MFC，是过滤程序还是服务器扩展程序，还是两者都有，以及在指定优先权下过滤什么通知，过滤程序是否确保将消息通知服务器。

注意： *ISAPI 扩展向导在 Visual Studio 2005 中就已经被移除了，所以 Visual Studio 2010 中也没有这个向导了，为了所讲知识的完整性，19.2.3~19.2.5 小节的内容将在 Visual C++ 6.0 上进行。*

在生成工程后，就可以增加自定义功能，并创建解析映射。其具体步骤如下。

（1）在 Visual C++ 6.0 开发环境下，选择 File|New 命令，打开 New 对话框，在左面的列表框中，选择 ISAPI Extension Wizard 选项，在右面的文本框中，分别输入工程名称、工程路径、工作区设置和平台信息。单击 OK 按钮，打开 ISAPI Extension Wizard–Step 1 of 1 对话框。

（2）在该对话框中，选择 Generate a Server Extension object 复选框，并在 Extension Class 文本框和 Extension 文本框中输入对应的类和扩展名，在下面的单选按钮组中选择使用何种方式链接到 MFC 中，单击 Finish 按钮，完成工程创建。

（3）现在可以在工程中添加要执行的功能，打开 MFC ClassWizard 对话框。在该对话框中可以看到向导为扩展类提供的消息，添加要处理的消息的处理函数。

（4）增加功能函数。当调用函数时，MFC 会传入一个 CHttpServerContext 对象的指针。读者也可以使用成员函数的回调函数获取附加的头信息，如用户的 IP 地址等信息。

（5）编译生成 DLL，并将其复制到支持 ISAPI 的 Web 服务器的运行目录下。如果是在 IIS 环境下，则需要将其放置在虚拟目录下，并且需要设置虚拟目录的执行权限。

（6）在客户端浏览器上输入对应的 URL，即可看到描述信息，输入指定的命令和参数，即可执行 ISA 的代码。下面是使用 ISAPI 增加用户登录记录的代码。

```
01  //增加用户
02  void CISASampleExtension::AddUser( CHttpServerContext* pctxt,
03      LPCTSTR pstrFirst, LPCTSTR pstrMiddle, LPCTSTR pstrLast )
04  {
05      try
06      {
```

```
07          CFile file;                                  //定义文件对象
08          file.Open("Users.txt", CFile::modeCreate|CFile::modeWrite);
09          //以写方式打开文件
10          CString log;                                 //定义日志变量
11          //格式化记录数据
12          log.Format("AddUser First=%s;Middle=%s;Last=%s",
13              pstrFirst, pstr          Middle, pstrLast);
14          file.Write(log, log.GetLength());            //将信息写入文件
15          file.Close();                                //关闭文件
16          AfxMessageBox("添加用户成功", MB_OK);        //显示提示消息框
17      }
18      catch(CException ex)                             //如果发生异常
19      {
20          ex.ReportError(MB_OK);                       //显示异常错误信息
21      }
22  }
```

当用户执行 ISA 的 AddUser()方法时，会将信息记录到 Users.txt 文件中。

19.2.4　调试 ISA

开发完 ISA 后，通常需要调试，才可以编写稳定的程序。调试 ISA 的步骤如下。

（1）启动 IIS 进程，在命令行下输入 net start iisadmin，或在控制面板中的 Internet 信息服务器中启动。

（2）在 Visual C++ 6.0 开发环境中，选择 Build|Start Debug|Attach to Process 命令，打开 Attache To Process 对话框，如图 19-4 所示。

（3）选择 Show System Process 复选框，从列表框中选择 inetinfo 进程，单击 OK 按钮。

（4）在命令行中输入 net start w3svc，或在服务组件中启动 World Wide Web Publishing Service。

（5）如果 Attache To Process 对话框的列表框中没有显示系统进程，则在服务组件中启动 IIS Admin，并选择“允许服务与桌面交互”复选框，如图 19-5 所示。

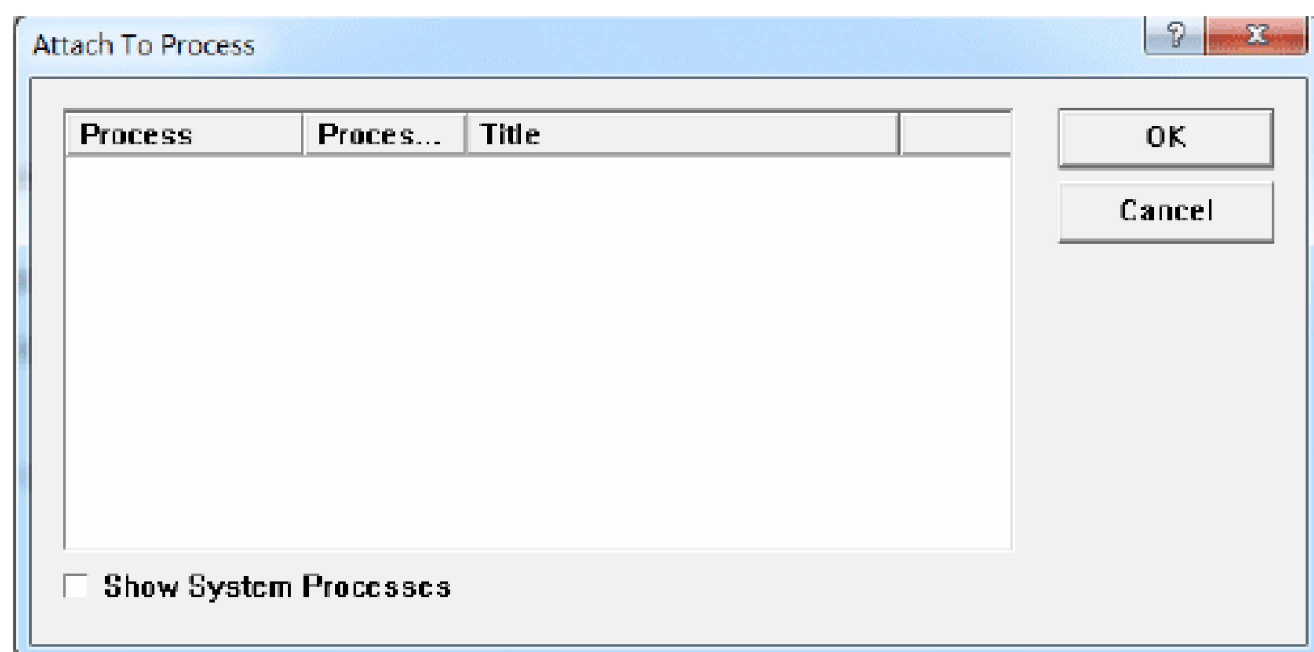


图 19-4　Attache To Process 对话框

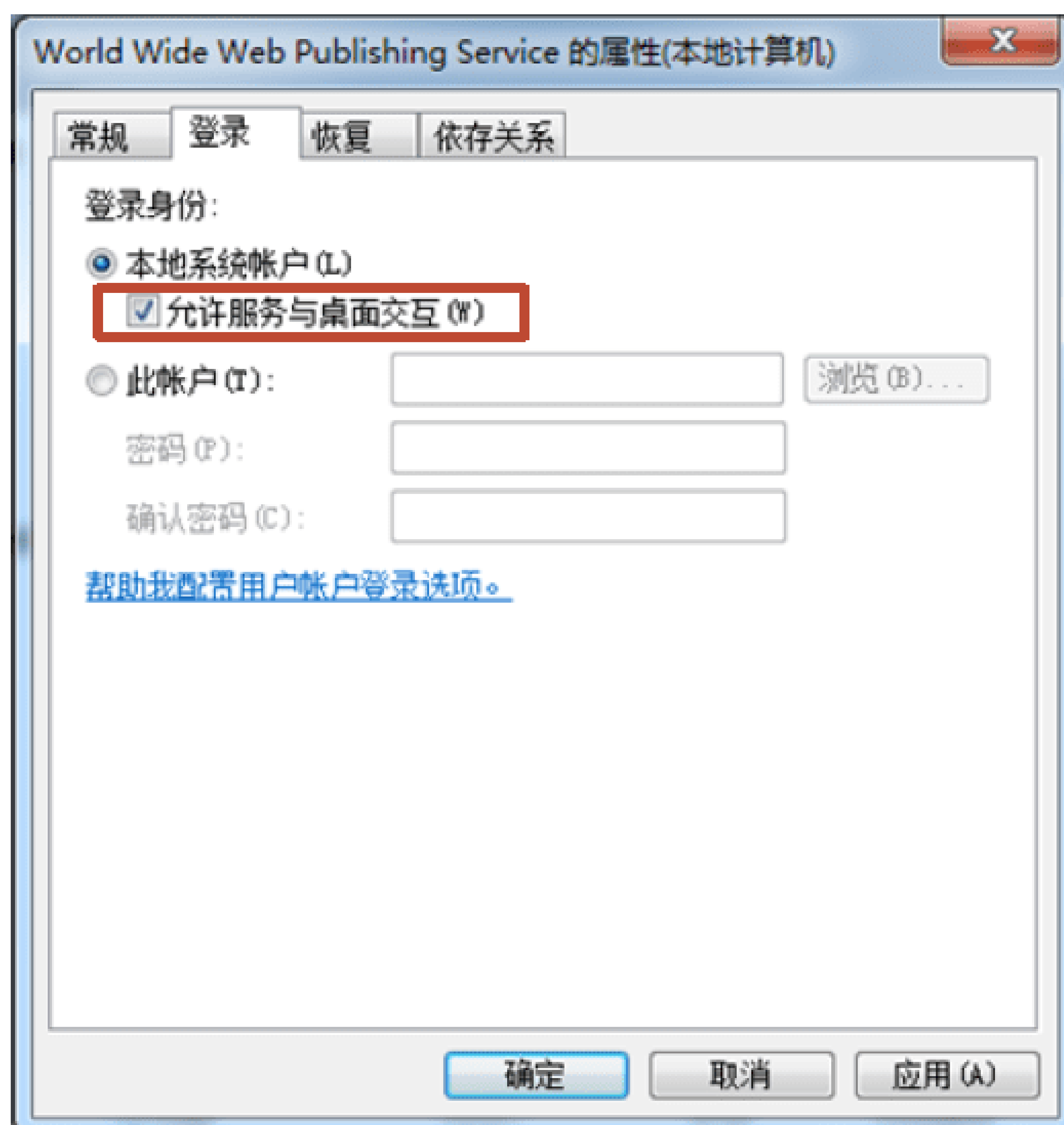


图 19-5　启动 IIS Admin 对话框

（6）重复步骤（4）和（5），将所有相关的服务的此选项都选上，即 WWW Publishing Service 服务和 FTP Publishing Service 服务。

（7）在命令行中输入 regedit 命令打开注册表编辑框，在注册表中增加 Inetinfo.Exe 项，并增加 Debugger 键值：

```
HKEY_LOCAL_MACHINE/Software/Microsoft/WindowsNT/CurrentVersion/Image
File Execution Options下
Debugger = <DebuggerExeName>
```

其中，DebuggerExeName 表示 Visual C++ 6.0 工具的完整路径。

（8）这样，就可以在 Visual C++ 6.0 中设置程序断点，运行并进行调试 ISA 了。

19.2.5　ISAPI 过滤程序

ISAPI 过滤程序是运行在支持 ISAPI 的 HTTP 服务器上的 DLL，用于过滤从服务器传入和传出的数据。过滤器注册通知事件，如记录日志或 URL 映射等。当注册的事件发生时，服务器会调用过滤器，使用过滤程序监控和修改发送给服务器或从服务器发送出去的数据。使用 ISAPI 过滤程序，可以增强 Internet 服务器定制特性，如增强 HTTP 请求记录功能、自定义加密和压缩方法或新的认证方法。

过滤程序是放置在客户端和 HTTP 服务器之间的。过滤程序可以在处理 HTTP 请求的任何阶段注册需要处理的事件，包括从客户端读取原数据时，使用 PCT 或 SSL 管理安全端口上的通信时，或处理 HTTP 请求的其他阶段。使用 MFC 的 CHttpFilter 类可以创建过滤程序，管理从 ISAPI 服务器出入的数据。该类包含的处理函数如表 19-5 所示。

表 19-5　CHttpFilter类的成员函数

函　数　名	功　　能
GetFilterVersion()	使用 CHttpFilter 需要将过滤程序的路径插入到 HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/W3SVC/Parameters/FilterDLLs 注册表中。当服务器启动时，读取此值并装载其中的 DLL。调用 CHttpFilter::GetFilterVersion()函数可以完成版本更新、确定请求的事件以及指定请求事件的优先级等功能
HttpFilterProc()	当事件发生时，服务器通知每个使用过滤程序的 HttpFilterProc()入口函数注册了事件的过滤程序。此函数用于确定处理哪些事件，并调用 ChttpFilter()成员函数处理，用户可以重载 HttpFilterProc()函数提供个性化的处理
OnPreprocHeaders()	当服务器预处理客户端信息头时的处理函数
OnAuthentication()	当验证客户端时使用的处理函数
OnUrlMap()	当服务器将逻辑 URL 映射到物理路径时的处理函数
OnSendRawData()	当服务器向客户端发送原数据时的处理函数
OnReadRawData()	当原数据从客户端发送给服务器之后，服务器处理之前，执行此处理函数
OnLog()	记录信息到服务器文件时的处理函数
OnEndOfNetSession()	当会话结束时使用的处理函数

使用 ISAPI 过滤程序可以实现多种服务器端的功能，如表 19-6 中列出了部分功能需要注册的过滤事件。

表 19-6　部分功能需要注册的过滤事件

要实现的功能	需要注册的过滤事件
统计访问服务器的用户数	SF_NOTIFY_LOG
自定义加密方式	SF_NOTIFY_READ_RAW_DATA、SF_NOTIFY_WRITE_RAW_DATA
自定义压缩方式	SF_NOTIFY_READ_RAW_DATA、SF_NOTIFY_WRITE_RAW_DATA
读取原数据	SF_NOTIFY_READ_RAW_DATA
处理头信息	调用 GetServerVariable 变量
认证用户	使用高优先权定义 SF_NOTIFY_AUTHENTICATION 事件
记录用户请求或关键字请求	SF_NOTIFY_URL_MAP

创建 ISAPI 过滤程序与创建 ISAPI 服务器扩展程序的步骤类似，不同之处如下所示。

（1）在 ISAPI Extension Wizard–Step 1 of 2 对话框中，选择 Generate a Filter object 复选框，并在 Filter Class 文本框和 Filter 文本框中输入过滤器类和过滤器名称。

（2）在上面的对话框中单击 Next 命令，打开 ISAPI Extension Wizard–Step 2 of 2 对话框，如图 19-6 所示。

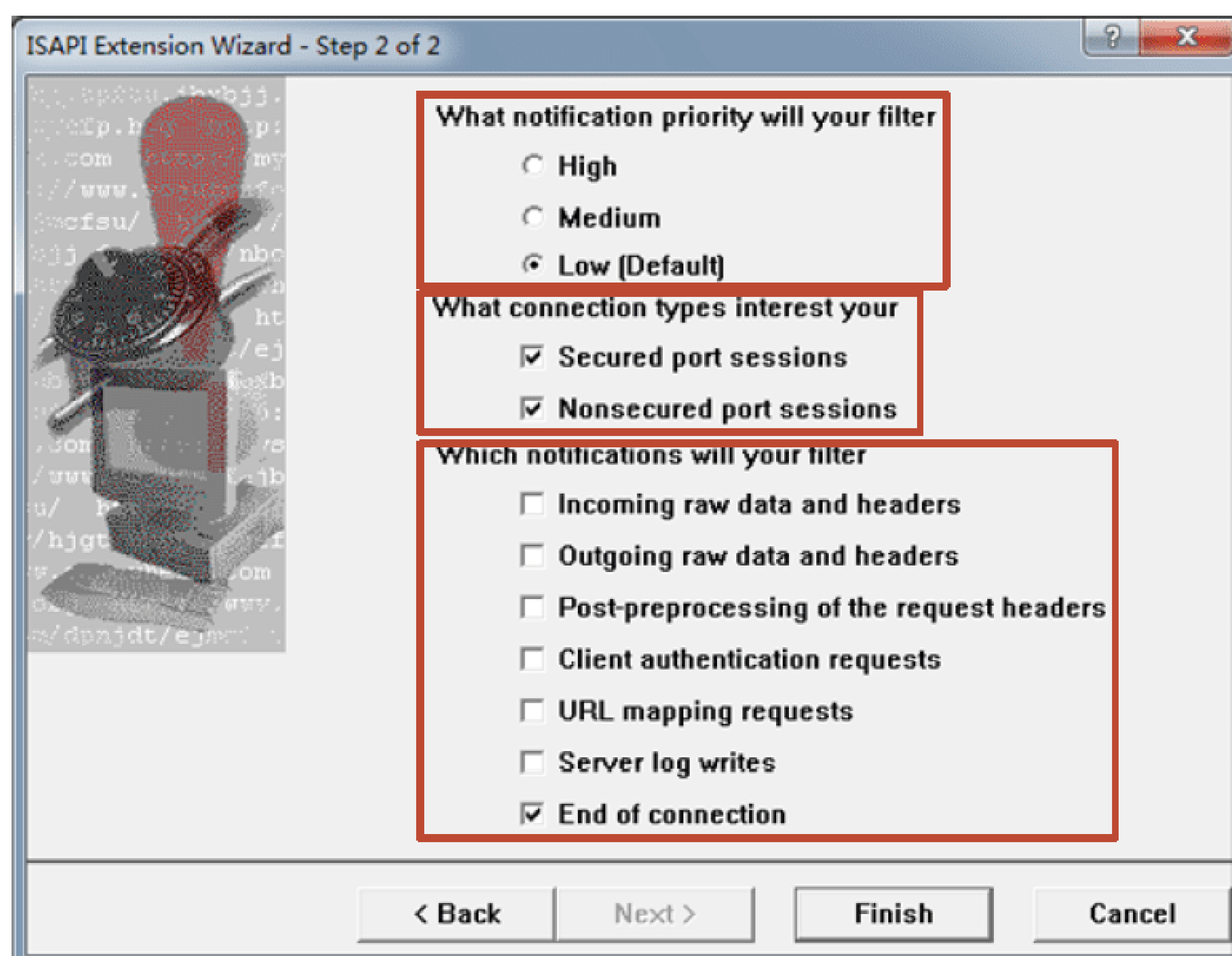


图 19-6　ISAPI Extension Wizard – Step 2 of 2 对话框

（3）在该对话框中，在 What notification priority will you filter 单选按钮组中选择过滤程序的优先权；在 What connection types interest your 复选框组中选择感兴趣的连接，包括安全连接和非安全连接；在最下面的复选框组中选择过滤程序要处理的事件。单击 Finish 按钮，完成工程的创建。

（4）在要处理的事件的处理函数中增加处理代码。以下代码会读取客户端的地址，并将其记录到 ISAPIFilterUserName.txt 文件中。

```
01  DWORD CISAPIFilterSampleFilter::OnLog(CHttpFilterContext *pCtxt,
02                                          PHTTP_ FILTER_LOG pLog)
03  {
04      //记录客户端信息
05      CFile file;                                            //定义文件变量
```

```
06     file.Open("C:\\ISAPIFilterUserName.txt",
07               CFile::modeCreate | CFile::modeWrite);//打开文件
08     //将客户端主机名称写入文件
09     file.Write(pLog->pszClientHostName,
10               strlen(pLog->pszClientHostName));
11     return SF_STATUS_REQ_NEXT_NOTIFICATION;     //继续调用下面的过滤器
12   }
```

（5）停止 World Wide Web 服务，复制 DLL 到指定目录中，将其添加到网站筛选器列表中，或在前面讲的注册表中加入此文件，重新启动服务即可。

19.3　MAPI 编程

MAPI（Messaging Application Programming Interface），即消息应用编程接口，可用于开发支持消息功能的应用程序。本节简单地介绍 MAPI 体系结构和 MAPI 应用程序接口，并以实例讲解如何在应用程序中增加对电子邮件功能的支持。

19.3.1　MAPI 体系结构概述

MAPI 是支持邮件功能的函数集，可以创建、管理和传输邮件消息，为开发人员提供定义邮件消息主题和内容的方法，并提供了灵活地存储邮件消息的方法。同时 MAPI 也为开发人员提供了一个创建不依赖于底层消息系统的带有邮件功能和邮件提醒功能的通用接口。消息接口提供了与 Windows 消息系统交互的人性化接口，并提供了与 MAPI 兼容的服务提供程序，MAPI 的体系结构如图 19-7 所示。

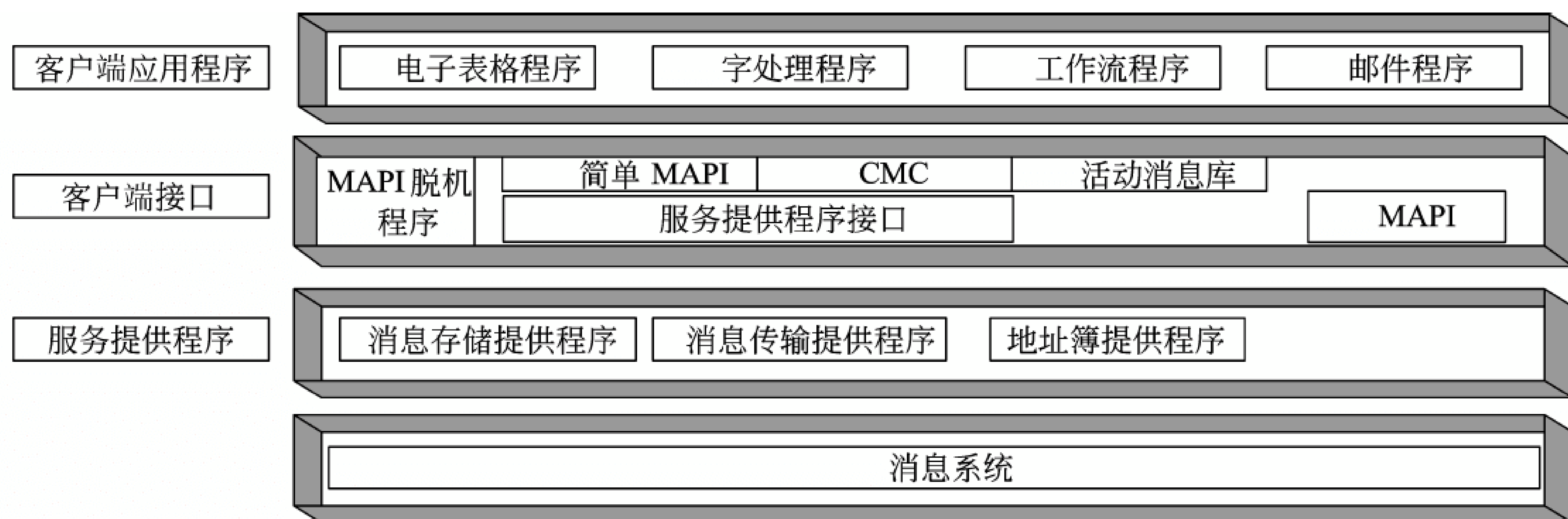


图 19-7　MAPI 体系结构

从图 19-7 中可以看出，体系结构的最上层是 MAPI 子系统的客户端应用程序，主要分为以下 3 种 MAPI 客户端应用程序。

- ❑ 消息通知应用程序：用于通知接收到消息的应用程序，如工作流应用程序，会通过通知消息的功能，提醒用户应该完成的工作。
- ❑ 消息发送应用程序：支持将程序中的部分内容以邮件的形式发送出去，如字处理程序和电子表格程序，都提供消息发送的功能，可以方便地将处理程序中的内容以邮件附件的方式发送给相关人员。

- ❑ 基于消息的应用程序：是将消息处理作为核心处理，并提供全面的消息特性的应用程序，可以完成各种格式的信息交换和保存。如电子邮件应用程序就是基于消息的应用程序。

MAPI 客户端程序需要与 MAPI 的客户端接口进行交互，由 MAPI 脱机程序、通用的用户接口和提供程序编程接口组成。MAPI 脱机程序是一个单独的进程，用于完成响应发送消息和从消息系统中接收消息的功能。通用用户接口是一组对话框，为客户端应用程序提供一个统一的界面，并为 MAPI 编程提供统一的工作方式。

MAPI 包含一组通用的编程接口，是使用服务提供程序提供的接口编写的，由 MAPI 客户端应用程序使用的一组通用的编程接口，称为 MAPI 应用程序接口。主要分为如下 3 种。

- ❑ 简单的 MAPI：是使用 C、VC 或 VB 编写的基于客户端应用程序接口的 API 函数。
- ❑ 通用消息调用（Common Messaging Calls，CMC）：是使用 C、C++编写的基于 API 函数的应用程序客户端接口。
- ❑ 活动消息库（Active Messaging Library）：是使用 C、C++、VB 或 VBA 编写的基于对象的应用程序客户端接口。

客户端应用程序可以使用上面这 3 种中的任何一种进行开发。这些接口使用起来非常简单。但是也略有不同，如需要的消息属性比较少的应用程序可以使用简单的 MAPI 和 CMC 编程接口，而如果要求支持的邮件功能速度快，则可以使用活动消息库。客户端应用程序不仅可以使用其中的任何一种接口，还可以组合使用这些接口，即在单个客户端应用程序中使用多种接口方式。

在 MAPI 体系结构中的接口层，还有一种接口是与服务提供程序进行交互的接口，是面向对象的 MAPI 编程接口的一部分。

服务提供程序放置在 MAPI 子系统和底层消息系统之间。从 MAPI 客户端应用程序发送出传输请求，服务提供程序会将其转换成消息系统特有的格式，并进行处理；当处理完成时，服务提供程序将其从消息系统特有的格式转换成 MAPI 格式，并通过接口传输给 MAPI 客户端应用程序。一般地，系统中有多种服务提供程序，用于处理不同的消息系统服务。如消息存储提供程序、消息传输提供程序和地址簿提供程序等。

在 MAPI 体系结构中的最后一层就是消息系统，它是 Windows 消息处理的底层机制。开发人员不需要关心细节处理，只要通过编程接口访问就可以了。

19.3.2　MAPI 应用程序接口

虽然 MAPI 提供的接口的种类有很多，API 函数也有多种，但是为了开发简便，MFC 在 CDocument 类中提供了一组 MAPI 子集，实现了对 MAPI 的部分封装，但是没有封装所有的 API。消息客户端提供了与 WMS（Windows Messaging System，Windows 消息系统）之间的人性化接口。这种交互包括从与 MAPI 兼容的服务提供程序处请求服务，如查询存储的消息和地址簿信息等。

MFC 封装了简单的 MAPI，可以使计算机上的邮件客户端很容易地将文档通过邮件附件发送出去。CDocument 具有确定在最终用户机器上是否支持邮件的成员函数，如果支持，则 ID_FILE_SEND_MAIL 命令是用于发送邮件的标准命令 ID。MFC 处理函数允许用户使用命令通过电子邮件发送文档。MFC 没有封装全部的 MAPI 函数集，读者可以直接调用

MAPI 函数，就像在 MFC 程序中调用其他 Win32 API 函数一样。

CDocument 类的 OnUpdateFileSendMail()函数可以检测当前机器上是否支持邮件功能。其函数原型为：

```
void OnUpdateFileSendMail( CCmdUI* pCmdUI );        //邮件发送接口更新命令
```

其中 pCmdUI 参数是指向与 ID_FILE_SEND_MAIL 命令相连的 CCmdUI 对象的指针。如果客户端上支持 MAPI，则会开启 ID_FILE_SEND_MAIL 命令。否则，函数会从菜单上移除 ID_FILE_SEND_MAIL 命令，包括与其相应的分隔符。判断是否支持 MAPI 的标准是在系统路径下是否具有 MAPI32.DLL 文件，并且在 WIN.INI 文件中 MAPI=1，则会认为系统是支持 MAPI 的。

CDocument 类的 OnFileSendMail()函数实现发送邮件的功能，其函数原型为：

```
void OnFileSendMail( );                             //发送邮件
```

OnFileSendMail()函数会将文档作为附件通过邮件主机发送出去。OnFileSendMail()函数调用 OnSaveDocument()函数将未保存的和修改的文档存为临时文件，然后将其通过电子邮件发送出去。如果没有装载 MAPI32.dll，则会装载此文件，因为要使用邮件功能，系统中必须包含此文件。

19.3.3　使用 MAPI 编写支持电子邮件的程序

19.3.2 小节介绍了 MAPI 的应用程序接口，本小节介绍如何使用 MFC 开发支持电子邮件功能的程序。首先创建支持 MAPI 的工程，创建方法与创建标准的文档/视图应用程序的步骤一样，只是在 “MFC 应用程序向导”对话框中，需要选择 MAPI（Messaging API）复选框，如图 19-8 所示。

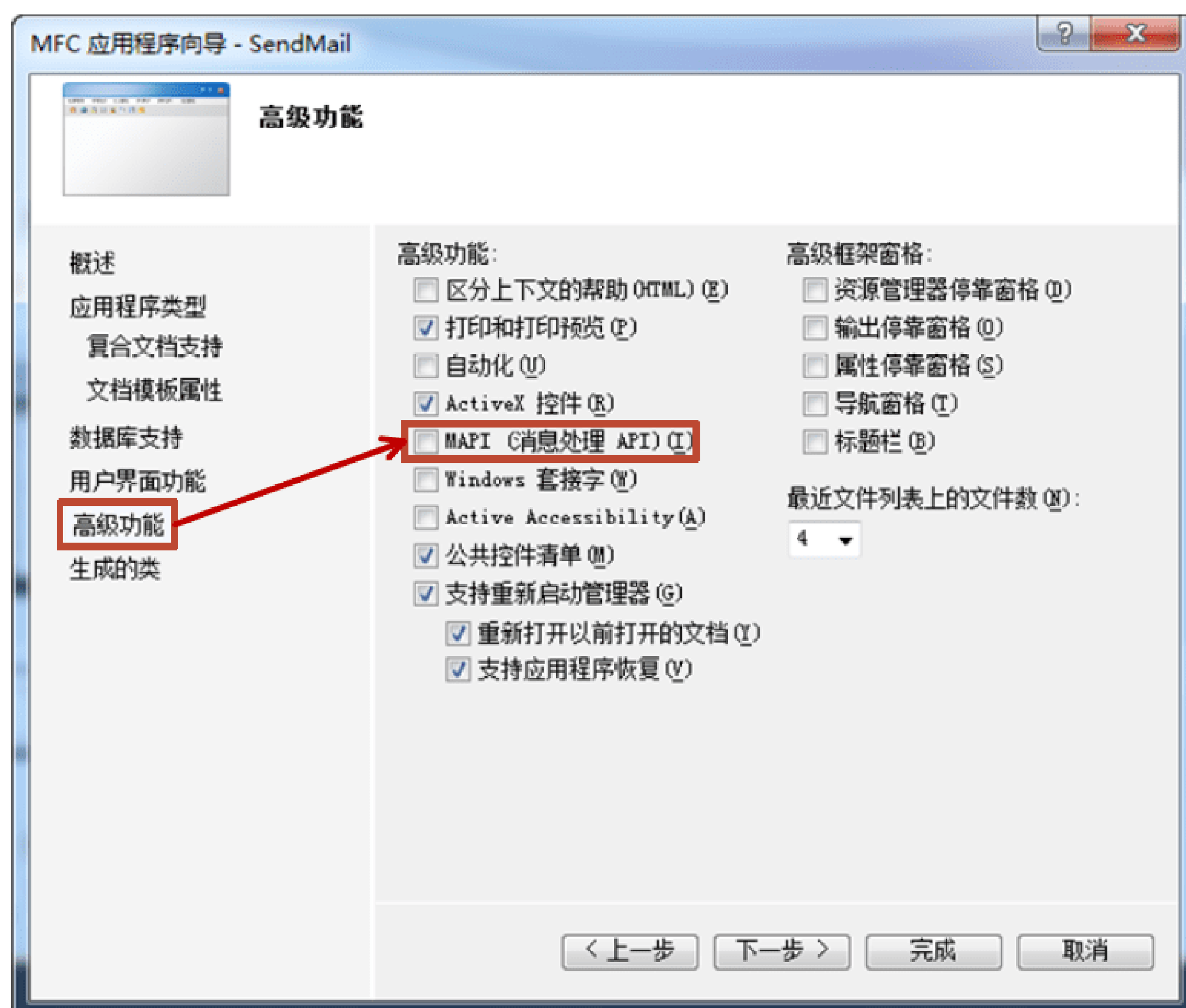


图 19-8　复选 MAPI 选项

这样创建的应用程序就自动将邮件发送功能集成进去了。主要在如下几个部分做了支持 MAPI 的修改。

（1）在菜单上增加名为 ID_FILE_SEND_MAIL 的菜单项，虽然可以增加在任何菜单上，但是一般将其增加在 File 菜单中。

（2）手动在文档的消息映射中增加如下部分：

```
01  BEGIN_MESSAGE_MAP(CMAPISampleDoc, CDocument)          //开始文档消息映射
02      ON_COMMAND(ID_FILE_SEND_MAIL, OnFileSendMail)  //邮件发送命令函数
03      ON_UPDATE_COMMAND_UI(ID_FILE_SEND_MAIL, OnUpdateFileSendMail)
04                                                        //邮件发送命令更新
05  END_MESSAGE_MAP()                                     //结束文档消息映射
```

（3）编译生成程序。如果系统支持邮件，则系统会显示“传送”菜单项，并使用 OnFileSendMail()函数处理命令，如果不支持邮件功能，则 MFC 会自动地移除“传送”菜单项，使用户看不到。程序运行的效果如图 19-9 所示。编辑完毕后，选择“文件”|“传送”命令，打开如图 19-10 所示的对话框。



图 19-9　MAPI 程序运行效果

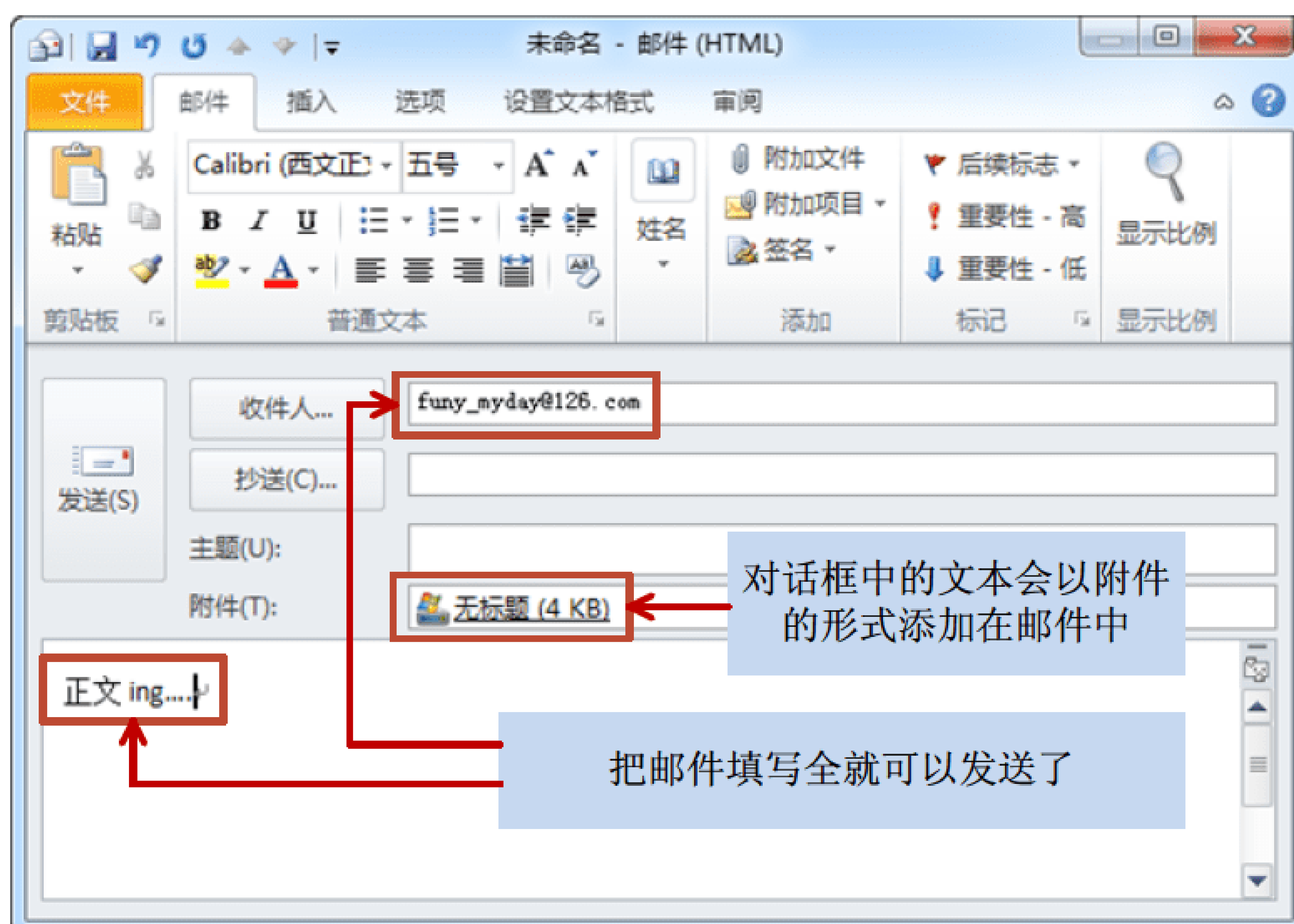


图 19-10　邮件发送窗体

在图 19-10 所示的对话框中，用户就可以像在 Windows 的邮件程序中发送和接收邮件的方式一样发送邮件了。要注意的是，如果读者没有在机器上设置邮件信息，则会打开如

图 19-11 所示的提示对话框。此时，读者需要到控制面板下的“邮件”选项中进行账号配置，才可使用邮件功能。在图 19-12 所示的对话框中，单击“添加”按钮，按照提示配置自己的邮件账号或地址簿即可。

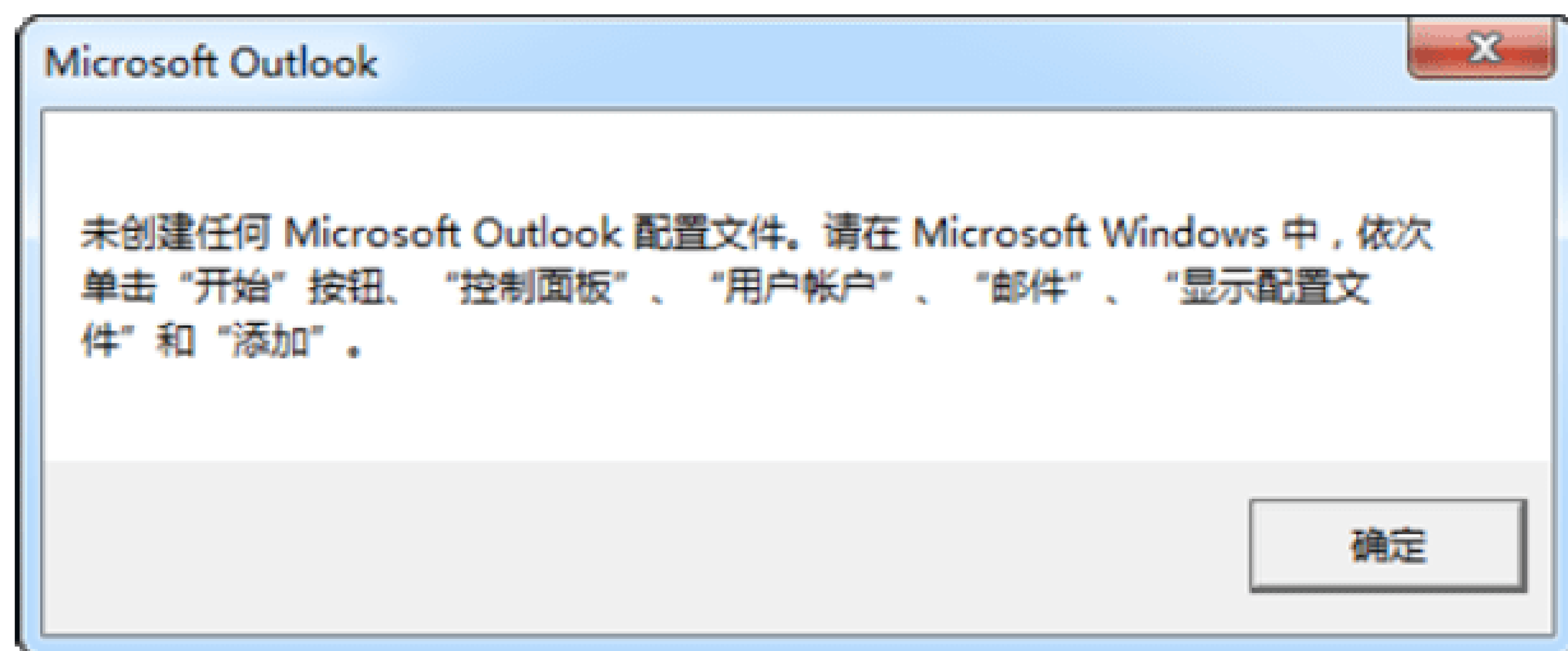


图 19-11 邮件配置提示

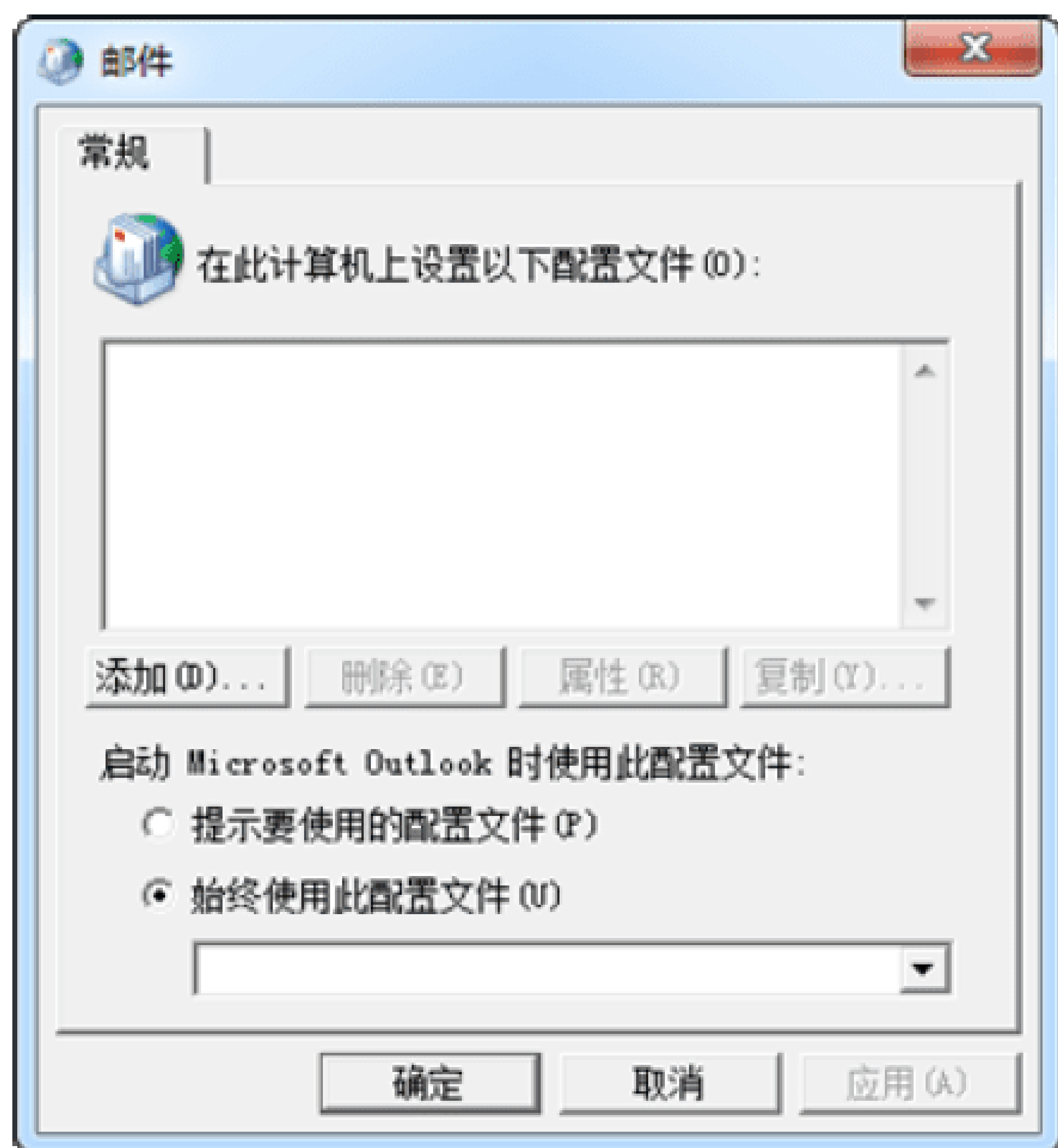


图 19-12 邮件配置对话框

19.4 本章小结

本章介绍了在 VC 中进行 Internet 编程的知识，包括 WinInet 编程、ISAPI 编程和 MAPI 编程。本章重点是掌握有关 Internet 编程的方法。本章的难点是深入理解这些 Internet 编程的底层原理。从第 20 章开始将介绍有关系统功能方面的编程。

19.5 习　　题

1．分别简要描述 WinInet API、ISAPI 和 MAPI 的作用。

【思路】参考 19.1 节的内容，对三者进行比较分析。

2．在 19.1.4 小节中的示例是基于单文档的。创建基于对话框的应用程序，实现相同的功能。对话框的界面设计可以是图 19-13 所示的样子。

【思路】将 19.1.4 小节中示例的代码移植到对话框中合适的位置。当然，可以使用到对话框的一些机制，比如 DDX 和 DDV 机制（参考第 10 章中的 10.2.3 小节）。

3．自己动手创建一个 Microsoft Outlook 配置文件。

【思路】参考 19.3.3 小节的内容。

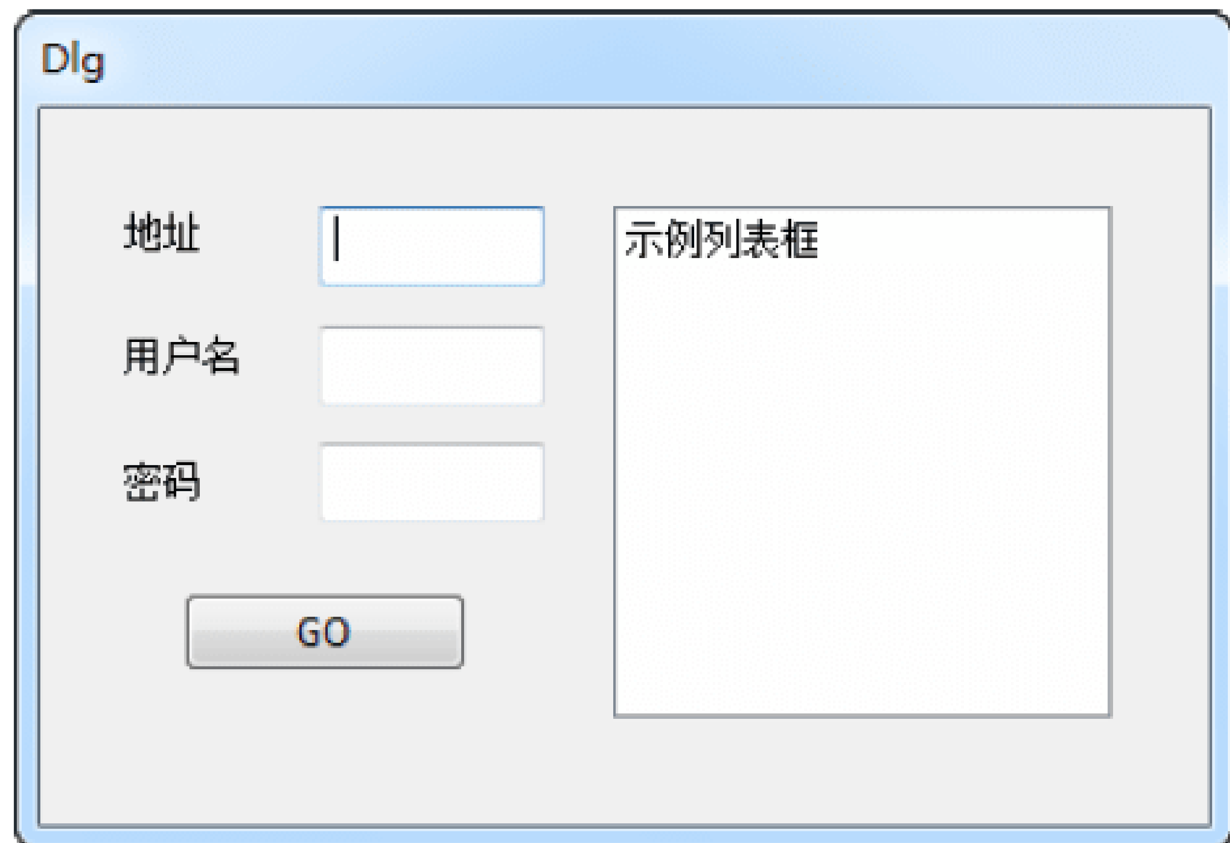


图 19-13 对话框界面设计

第 5 篇　系统编程

第 20 章　系统相关功能开发

本章主要介绍使用 Visual Studio 2010 开发与 Windows 系统相关功能的知识。包括如何获取磁盘信息、如何操作磁盘、如何执行系统控制与调用、如何执行应用程序的操作、如何执行系统工具、如何执行与桌面相关的操作、如何获取系统信息、Windows 消息的使用、剪贴板的使用以及与鼠标键盘相关的操作。

20.1　获取磁盘信息

Windows 操作系统中，提供了一组控制输入、输出设备的 API 函数。应用程序使用这些函数可以直接与设备驱动进行通信，还可以完成输入和输出操作，或者返回有关诸如软盘驱动器、硬盘驱动器、磁带驱动器或 CD-ROM 驱动器等的信息。

20.1.1　获取驱动器卷标

通过调用 GetVolumeInformation()函数可以获取驱动器的卷标。其函数原型为：

```
BOOL GetVolumeInformation(                //获取驱动器的卷标
  LPCTSTR lpRootPathName,                 //要获取卷标信息的驱动器的根目录
  LPTSTR lpVolumeNameBuffer,              //存放获取卷标信息的缓冲区的指针
  DWORD nVolumeNameSize,                  //存放获取卷标信息的缓冲区的大小
  LPDWORD lpVolumeSerialNumber,           //存放获取的盘卷序列号的缓冲区指针
  LPDWORD lpMaximumComponentLength,//最大的文件名长度
  LPDWORD lpFileSystemFlags,              //指向与文件系统相关的标记的组合的指针
  LPTSTR lpFileSystemNameBuffer,          //存放获取的文件系统名称的缓冲区的指针
  DWORD nFileSystemNameSize);             //指定存放获取的文件系统名称的缓冲区的大小
```

其中，参数 lpFileSystemFlags 是指向与文件系统相关的标记的组合的指针。表 20-1 中列出了有效的标记及其含义，其中各项可依据系统任意进行组合，但是 FS_FILE_COMPRESSION 选项和 FS_VOL_IS_COMPRESSED 选项是互斥的。

表 20-1　文件系统标记

取　　值	含　　义
FS_CASE_IS_PRESERVED	文件名的大小写保存在文件系统中
FS_CASE_SENSITIVE	文件系统中对文件名是区别大小写的
FS_UNICODE_STORED_ON_DISK	文件系统支持 Unicode 文件名
FS_PERSISTENT_ACLS	文件系统支持文件的访问控制列表安全机制，如 NTFS 文件系统支持，而 FAT 文件系统不支持

续表

取　值	含　义
FS_FILE_COMPRESSION	文件系统支持基于文件的压缩。如果文件系统支持，则单个此文件系统下的文件可以压缩，也可以不压缩
FS_VOL_IS_COMPRESSED	指定的卷是个压缩卷
FILE_SUPPORTS_ENCRYPTION	文件系统支持加密的文件系统（EFS）
FILE_SUPPORTS_OBJECT_IDS	文件系统支持对象标识
FILE_SUPPORTS_REPARSE_POINTS	文件系统支持 Reparse Point
FILE_SUPPORTS_SPARSE_FILES	文件系统支持稀疏文件
FILE_VOLUME_QUOTAS	文件系统支持磁盘卷引用

如果使用此函数获取软驱或光驱的信息，但是软驱中没有软盘或者光驱中没有光盘，则系统会弹出消息框，提示用户插入软盘或光盘。要阻止系统显示此消息框，可以使用 SEM_FAILCRITICALERRORS 参数调用 SetErrorMode()函数。具体代码如下：

```
01  void CDiskInfoDlg::OnButtonGetvol()//获取驱动器卷标
02  {
03      UpdateData(true);             //从控件中更新数据，更新要获取的驱动器名称
04      char szVolume[MAX_PATH]={0};     //存放卷标信息的字符数组
05      if (GetVolumeInformation(m_DiskName, szVolume, MAX_PATH,NULL,
06           NULL,NULL,NULL, 0))
07      {
08          //获取卷标
09          WriteLog("驱动器"+ m_DiskName + "的卷标=" + szVolume);//记录日志
10      }
11      else                             //获取卷标失败
12      {                                //检测是否没有插入盘
13          if (GetLastError() == ERROR_DEVICE_NOT_CONNECTED)
14              WriteLog("没有插入盘");
15          else
16              WriteLog("获取驱动器卷标失败");//显示错误提示
17      }
18  }
```

上面函数调用 GetVolumeInformation()函数获取“磁盘”文本框中指定的驱动器的卷标，并在提示文本框中显示操作结果。程序运行效果如图 20-1 所示。

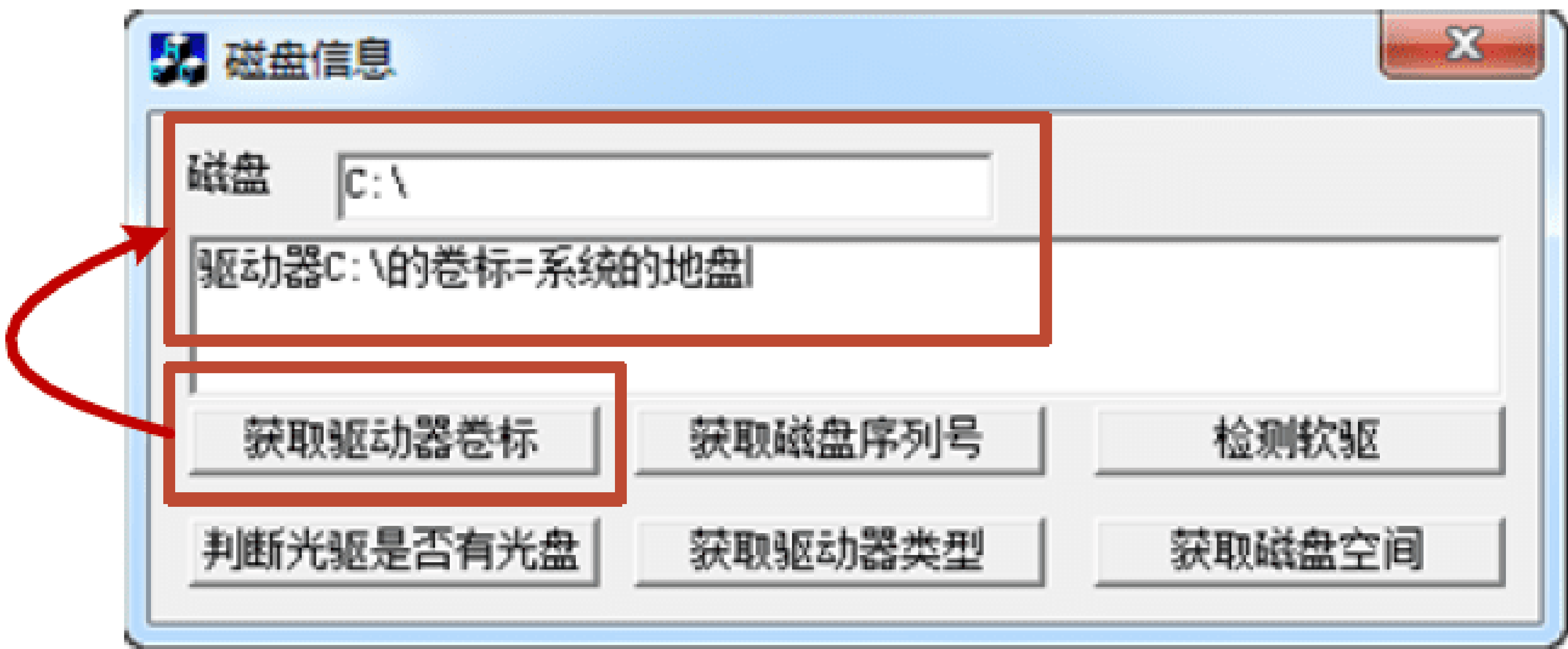


图 20-1　获取驱动器卷标运行效果

20.1.2　获取磁盘序列号

获取磁盘序列号与获取驱动器卷标使用的函数是相同的，GetVolumeInformation()函数

的使用如 20.1.1 小节所述。不同之处在于使用的参数不同，驱动器卷标是存放在函数的第二个参数指定的缓冲区中，而磁盘序列号是存放在函数的第四个参数指定的缓冲区中。以下代码显示了如何获取磁盘序列号。

```
01  void CDiskInfoDlg::OnButtonGetserial() //获取磁盘序列号
02  {
03      UpdateData(true);             //从控件中更新数据，更新要获取的驱动器名称
04      DWORD dwSerial;                   //存放磁盘序列号的字符数组
05      if (GetVolumeInformation(m_DiskName,NULL,0,&dwSerial,NULL,
06                    NULL,NULL,0))       //获取磁盘序列号
07          WriteLog("磁盘%s 的磁盘序列号=%X", m_DiskName, dwSerial);
08                                        //显示获取的磁盘序列号
09      else
10          WriteLog("获取驱动器卷标失败");//显示错误提示
11  }
```

上面函数调用 GetVolumeInformation()函数获取磁盘序列号，并在日志文本框中显示获取的磁盘序列号。程序运行效果如图 20-2 所示。



图 20-2　获取磁盘序列号运行效果

20.1.3　检测软驱是否有软盘

DeviceIoControl()函数提供控制设备输入和输出的接口，使用此函数，程序可以向设备发送各种控制代码。通过传入 IOCTL_STORAGE_CHECK_VERIFY 设备控制代码调用此函数可以检测软驱中是否有软盘。此函数原型为：

```
BOOL DeviceIoControl(
  (HANDLE) hDevice,                 //要操作的设备的句柄
  IOCTL_STORAGE_CHECK_VERIFY,       //要完成的控制代码，此处指检验存储设备
  NULL,                             //输入缓冲区指针，此处必须为 NULL
  0,                                //输入缓冲区的大小，此处必须为 0
  NULL,                             //输出缓冲区指针，此处必须为 NULL
  0,                                //输出缓冲区的大小，此处必须为 0
  (LPDWORD) lpBytesReturned,        //存放输出字节数的指针变量，表示返回的字节数
  (LPOVERLAPPED) lpOverlapped);     //存放异步操作的 OVERLAPPED 结构的指针
```

下面的代码演示了如何使用此函数检测软驱中是否有软盘。

```
01  void CDiskInfoDlg::OnButtonCheckA()     //检测软驱是否有软盘
02  {
```

```
03      BOOL bResult = false;                //操作结果变量
04      DWORD cb = 0;                        //输出字节数，此处无实际意义
05      HANDLE hDevice = NULL;               //设备句柄
06      hDevice = CreateFile("\\\\.\\A:\\", GENERIC READ, 0, 0,
07        OPEN EXISTING,FILE SHARE WRITE, NULL);//打开到软驱的连接句柄
08      if (hDevice != NULL)                 //如果打开句柄成功
09      {
10          //检测软驱中是否有软盘
11          if (DeviceIoControl((HANDLE) hDevice,
12               IOCTL STORAGE CHECK VERIFY, NULL, 0, NULL, 0,&cb, NULL))
13          {
14              WriteLog("软驱中有软盘");       //输出信息提示
15              return;                      //返回
16          }
17      }
18      WriteLog("软驱中没有软盘");              //输出信息提示
19  }
```

上面代码先使用 CreateFile()函数打开到软驱 A 的连接，并存储句柄，接着调用 DeviceIoControl()函数检测其中是否有软盘，并输出操作结果。程序运行效果如图 20-3 所示。



图 20-3　检测软驱中是否有软盘的运行效果

20.1.4　判断是否插入存储器

在 Windows 平台下，通过 WM_DEVIC-ECHANGE 消息可以检测是否插入存储器以及存储器是否被安全拔出。此消息传入的第一个参数 wParam 表示设备改变的类型，如果此参数为 DBT_DEVICEARRIVAL，表示有新硬件插入；如果此参数为 DBT_DEVICER-EMOVECOMPLETE，表示硬件成功移除。此消息传入的第二个参数 lParam 表示 PDEV_BROADCAST_HDR 结构的设备信息。下面是判断是否插入存储器的代码。

```
01  LRESULT CDiskInfoDlg::WindowProc(UINT message, WPARAM wParam,
02                                      LPARAM lParam)
03  {
04      //处理 Windows 消息
05      if (message == WM_DEVICECHANGE)          //如果消息是有设备改变
06          HandleDeviceChange(message, wParam, lParam);
07                                               //调用处理设备改变消息函数
08      return CDialog::WindowProc(message, wParam, lParam);
09  }
```

上面代码是检测消息事件的函数，如果消息为设备改变消息 WM_DEVICECHANGE，则会调用 HandleDeviceChange()函数，处理代码如下：

```
01  void  CDiskInfoDlg::HandleDeviceChange(UINT message,WPARAM wParam,
02                                         LPARAM lParam)
03  {
04      //检测驱动器插入和移除
05      PDEV_BROADCAST_HDR lpdb=(PDEV_BROADCAST_HDR)lParam;
06      //获取改变的设备信息
07      switch(wParam)                                //判断消息类型
08      {
09      case  DBT_DEVICEARRIVAL:                      //插入新硬件
10          //如果设备是存储器
11          if   (lpdb->dbch_devicetype == DBT_DEVTYP_VOLUME)
12          {
13              //转换设备信息
14              PDEV_BROADCAST_VOLUME lpdbv = (PDEV_BROADCAST_VOLUME)lpdb;
15              if (!((lpdbv->dbcv_flags & DBTF_MEDIA)  ||
16                      (lpdbv->dbcv_flags &DBTF_NET)))
17              {
18                  //如果设备为不为媒体和网络设备，则获取存储器卷标
19                  //获取卷标
20                  char volumn = GetDriveFromMask(lpdbv->dbcv_unitmask);
21                  WriteLog("插入%c 新存储器", volumn);//显示消息
22              }
23          }
24          break;
25      case   DBT_DEVICEREMOVECOMPLETE:              //完成删除硬件
26          //如果设备是存储器
27          if  (lpdb->dbch_devicetype == DBT_DEVTYP_VOLUME)
28          {
29              //转换设备信息
30              PDEV_BROADCAST_VOLUME lpdbv = (PDEV_BROADCAST_VOLUME)lpdb;
31              if (!((lpdbv->dbcv_flags & DBTF_MEDIA)  ||
32                          (lpdbv->dbcv_flags &DBTF_NET)))
33              {
34                  //如果设备不为媒体和网络设备，则获取存储器卷标
35                  char volumn = GetDriveFromMask(lpdbv->dbcv_unitmask);
36                  //获取卷标
37                  WriteLog("移除%c 存储器", volumn);  //显示消息
38              }
39          }
40          break;
41      default:                                      //默认
42          break;                                    //不做处理，退出
43      }
44  }
```

上面代码判断了消息参数是插入设备还是移除设备，并判断设备是否是存储器。如果是存储器，则会显示系统为存储器分配的盘符。获取盘符的函数代码如下：

```
01  //获取驱动器的卷标
02  char CDiskInfoDlg::GetDriveFromMask (ULONG unitmask)
03  {
04      char i;                                 //定义字符变量 i
05      for (i = 0; i < 26; ++i)                //依次循环判断卷标
06      {
```

```
07          if (unitmask & 0x1)
08              break;                              //如果标志位置位，则退出循环
09          unitmask = unitmask >> 1;               //否则，右移一位，继续循环
10      }
11      return (i + 'A');                           //返回字母卷标
12  }
```

上面的函数使用传入的掩码判断其第一位的值，并计算相对于字符 A 的值，即系统为其分配的盘符。上面的示例运行后，当插入新的存储器或移除存储器后，会在信息文本框中显示提示信息，程序运行效果如图 20-4 所示。

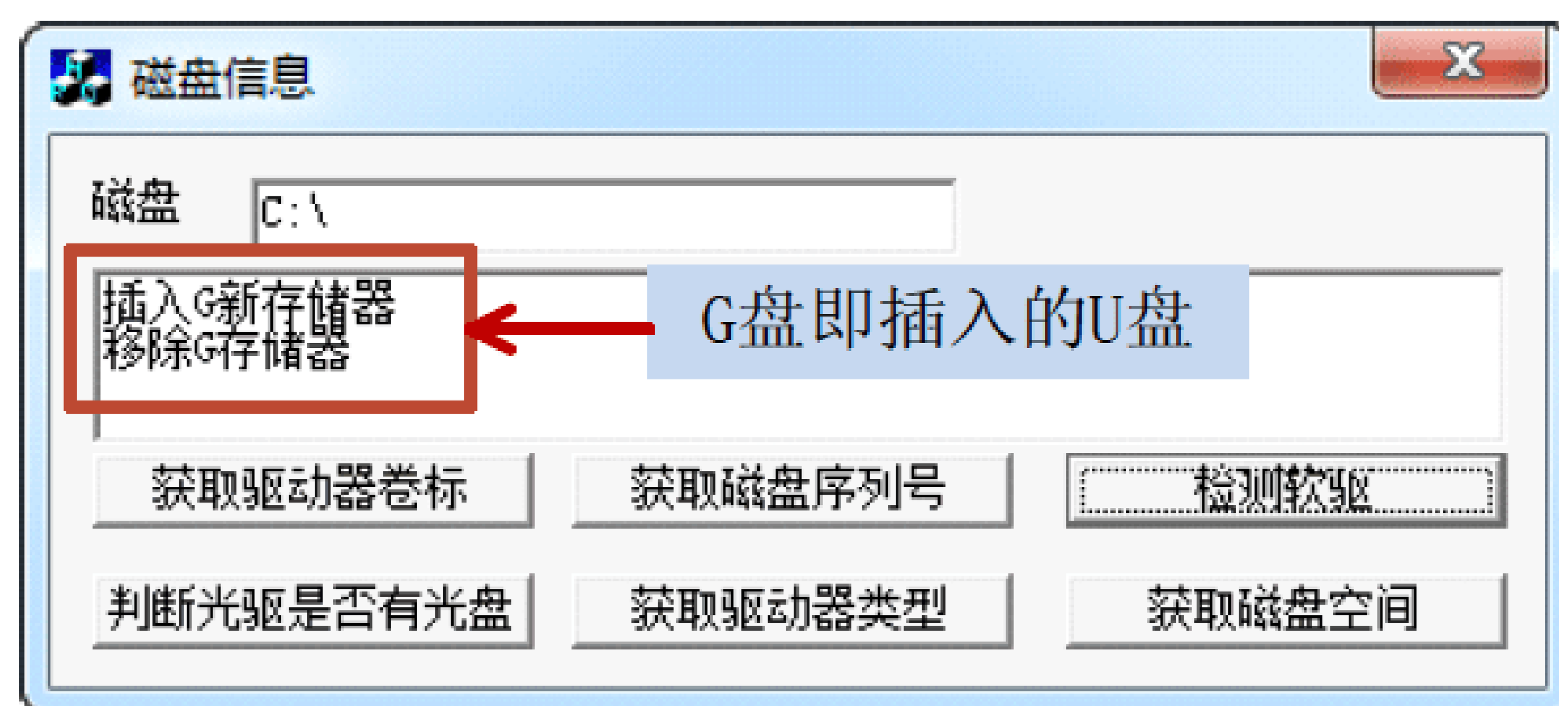


图 20-4　判断是否插入存储器运行效果

20.1.5　判断光驱是否有光盘

判断光驱是否有光盘，是根据 GetVolumeInformation()函数的返回值进行判断。如果返回值为 0，表示其中没有光驱，因此无法获取有效的光盘信息；如果返回值为其他值，表示光驱中含有光盘信息。下面的代码演示了判断光驱中是否有光盘的方法。

```
01  void CDiskInfoDlg::OnButtonCheckcdrom()         //检测是否有光盘
02  {
03      DWORD dwReturn;                             //定义返回值变量
04      char strDrivers[MAX_PATH];                  //定义驱动器数组
05      dwReturn = GetLogicalDriveStrings(MAX_PATH, (LPSTR)&strDrivers);
06      //获取本地驱动器字符串
07      CString log;                                //定义日志信息变量
08      for (int i = 0;i < dwReturn; i++)           //循环判断驱动器的情况
09      {
10          //如果驱动器盘符是有效的
11          if ((strDrivers[i] <= 'Z') && (strDrivers[i] >= 'A'))
12          {
13              CString driver;                     //定义驱动器字符串
14              driver.Format("%c:\\", strDrivers[i]);//格式化字符串
15              UINT type = GetDriveType(driver);//获取指定盘符驱动器的类型
16              if (type == DRIVE_CDROM)            //如果是光驱
17              {
18                  int bResult = GetVolumeInformation(driver, NULL, 0,
19                      NULL, NULL, NULL, NULL, 0); //获取卷标信息
20                  CString info;               //定义存放卷标的字符串变量
21                  if (bResult == 0)           //如果返回的结果为 0，则没有光盘
22                      info.Format("光驱%c 中没有光盘\r\n", strDrivers[i]);
23                  else                        //否则，光驱中有光盘
```

```
24                  info.Format("光驱%c 中有光盘\r\n", strDrivers[i]);
25              log += info;                //添加日志提示信息
26          }
27      }
28    }
29    WriteLog(log);                        //显示操作结果
30 }
```

上面代码首先使用 GetLogicalDriveStrings()函数获取当前系统中所有的驱动器盘符，遍历这些驱动器，判断每个驱动器的类型是否是光驱 DRIVE_CDROM。如果是光驱，则调用 GetVolumeInformation()函数获取驱动器的信息。如果返回值为 0，表示当前光驱中没有光盘；否则，表示当前光驱中有光盘。程序运行效果如图 20-5 所示。

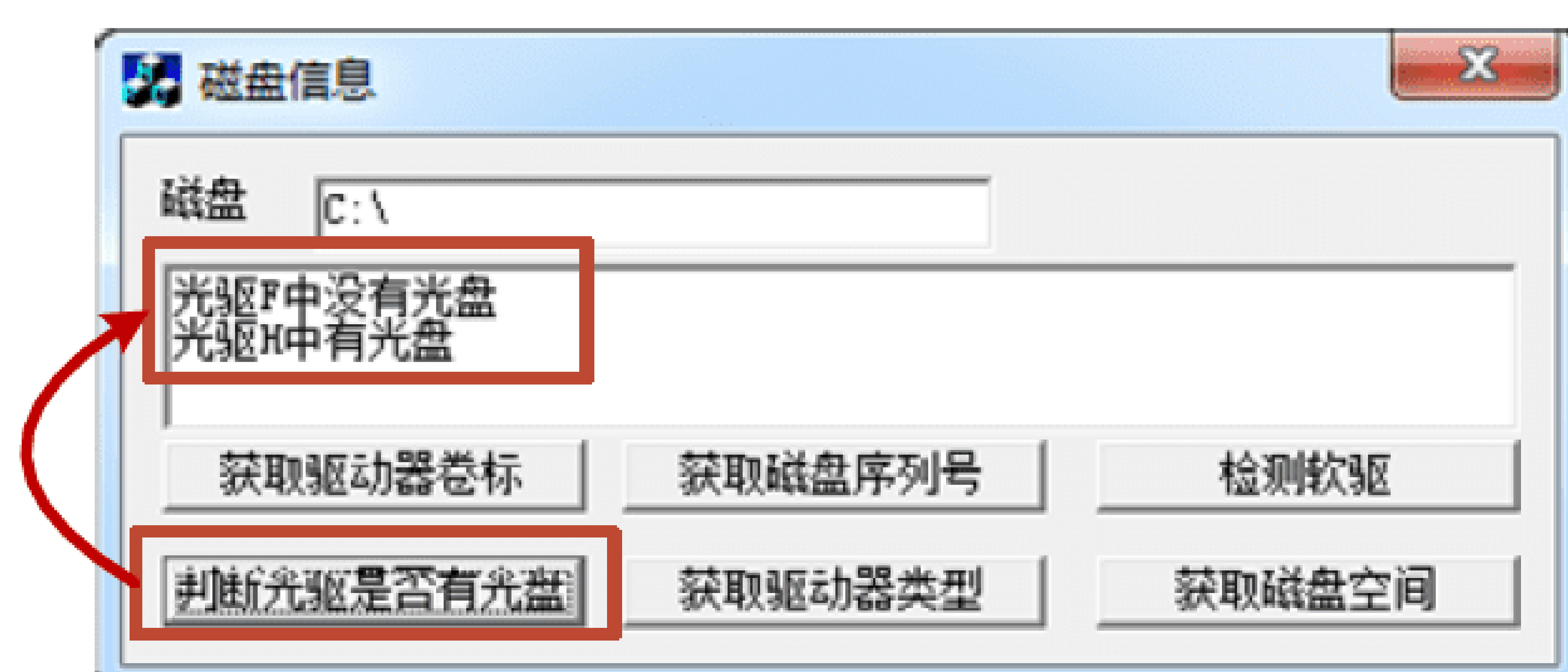


图 20-5　判断光驱中是否含有光盘运行效果

20.1.6　判断驱动器类型

调用 Win32 API 函数 GetDriverType()可以判断驱动器类型，其函数原型为：

```
UINT GetDriveType(
  LPCTSTR lpRootPathName);          //指向驱动器根路径的指针，如 C:\
```

此函数的返回值为 UINT 类型的数据，表示指定的驱动器的类型。表 20-2 列出了各个返回值代表的驱动器类型。

表 20-2　驱动器取值及类型

取　　值	代表的驱动器类型
DRIVE_UNKNOWN	驱动器类型不能确定
DRIVE_NO_ROOT_DIR	指定的根目录不存在
DRIVE_REMOVABLE	可移动的存储器
DRIVE_FIXED	固定的驱动器，也就是通常所说的磁盘
DRIVE_REMOTE	远程驱动器，也称为网络驱动器
DRIVE_CDROM	CD-ROM 驱动器
DRIVE_RAMDISK	内存驱动器，此种类型的驱动器是将内存的一部分划分出来当作硬盘使用

下面代码演示了如何使用 GetDriveType()函数获取驱动器类型。

```
01  void CDiskInfoDlg::OnButtonGetmediatype()
02  //判断驱动器类型
03  {
```

```
04      UpdateData(true);              //从控件中更新数据，更新要获取的驱动器名称
05      CString  csType;                  //存放驱动器类型的字符串
06      UINT uiType = GetDriveType(m_DiskName); //获取驱动器类型
07      switch (uiType)                   //根据返回值判断驱动器类型
08      {
09      case DRIVE_UNKNOWN:
10          csType = "驱动器类型不能确定。";    break;
11      case DRIVE_NO_ROOT_DIR:
12          csType = "指定的根目录不存在。";    break;
13      case DRIVE_REMOVABLE:
14          csType = "可移动的存储器。";   break;
15      case DRIVE_FIXED:
16          csType = "固定的驱动器，也就是通常所说的磁盘。";  break;
17      case DRIVE_REMOTE:
18          csType = "远程驱动器，也称为网络驱动器。";   break;
19      case DRIVE_CDROM:
20          csType = "CD-ROM 驱动器"; break;
21      case DRIVE_RAMDISK:
22          csType = "内存驱动器，驱动器将内存的一部分划分出来当作硬盘使用。";
23          break;
24      default:
25          csType = "未知";
26      }
27      //显示操作信息
28      WriteLog("驱动器%s 的类型返回值=%d（%s）",m_DiskName,uiType,csType);
29  }
```

上面代码调用 GetDriveType()函数获取指定驱动器的类型，并根据返回值，返回其类型文字描述。在实际操作中，用户可以根据自己的需求和驱动器类型，执行相应的操作。程序运行效果如图 20-6 所示。

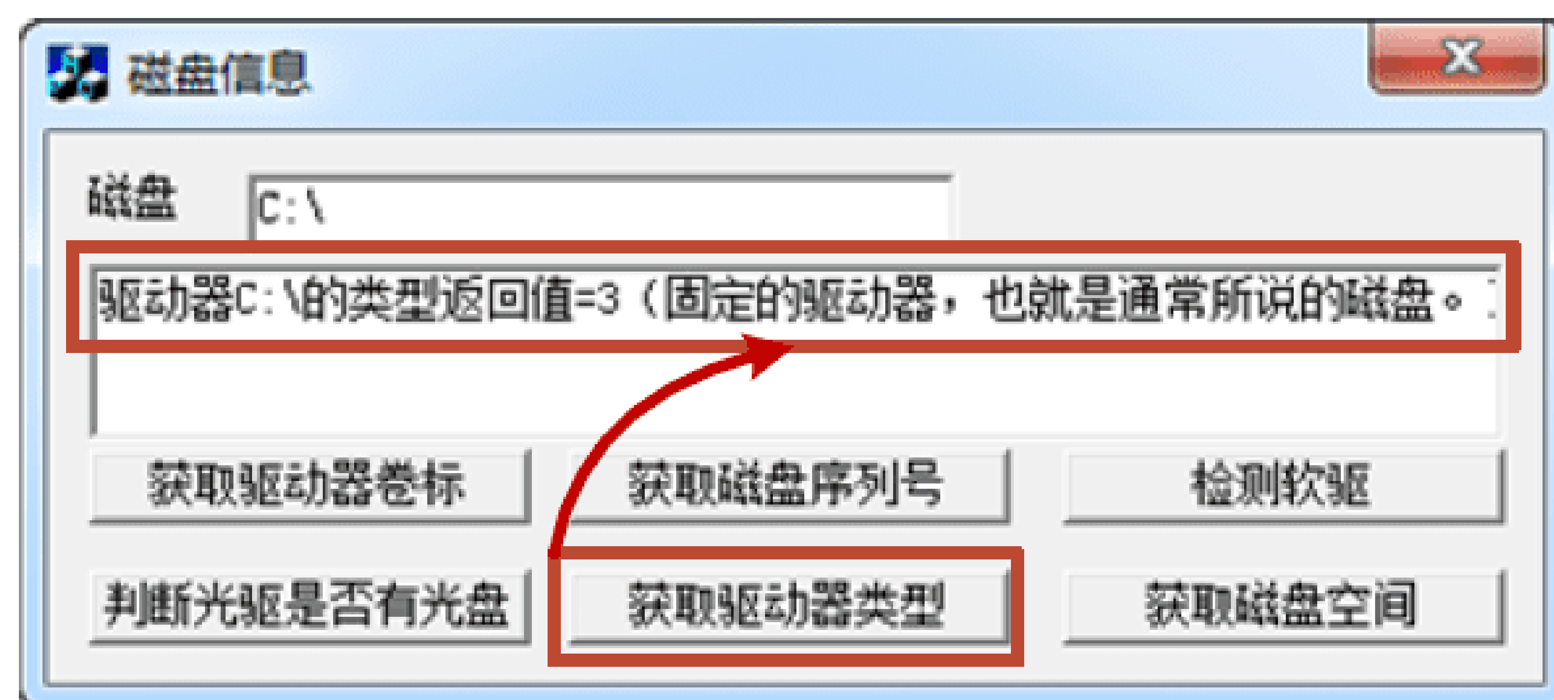


图 20-6　判断驱动器类型运行效果

20.1.7　获取磁盘空间信息

通过调用 Win32 API 函数 GetDiskFreeSpace()和 GetDiskFreeSpaceEx()可以获取指定磁盘的信息，其中包括磁盘的可用空间信息。其中，GetDiskFreeSpaceEx()函数是 GetDiskFreeSpace()函数的升级，因此，基于 Win32 的函数应该使用 GetDiskFreeSpaceEx()函数获取有关磁盘的信息。其函数原型为：

```
BOOL GetDiskFreeSpace(
  LPCTSTR lpRootPathName,            //指向驱动器根路径的指针
  LPDWORD lpSectorsPerCluster,   //DWORD 类型的指针，其中存储每柱面包含的扇区数
  LPDWORD lpBytesPerSector,      //DWORD 类型的指针，其中存储每扇区包含的字节数
```

```
  LPDWORD lpNumberOfFreeClusters,   //DWORD 类型的指针，其中存储空闲柱面数
  LPDWORD lpTotalNumberOfClusters);//DWORD 类型的指针，其中存储总柱面数
BOOL GetDiskFreeSpaceEx(            //获取空闲磁盘空间信息的扩展函数
  LPCTSTR lpDirectoryName,          //指向要获取信息的目录名称的指针
  PULARGE INTEGER lpFreeBytesAvailableToCaller,
                                    //对调用者可用的磁盘字节数
  PULARGE_INTEGER lpTotalNumberOfBytes,         //磁盘的总字节数
  PULARGE_INTEGER lpTotalNumberOfFreeBytes);    //磁盘的空闲字节数
```

GetDiskFreeSpaceEx()函数可以获取磁盘上可用空间的信息，包括磁盘空间的大小、空闲磁盘的大小，以及与调用线程相关的用户可用的空闲磁盘大小。在使用此函数时，要注意参数类型为 ULARGE_INTEGER，此类型是由两个 32 位的整型组成，即 64 位整型。

如果在早期的 Windows 版本中，要实现此功能，则需要确认系统中是否支持 GetDiskFreeSpaceEx()函数。如果不支持，则使用 GetDiskFreeSpace()函数实现。至于如何判断系统是否支持此函数，可以动态加载 KERNEL32.DLL 动态链接库，通过调用 GetProcAddress()函数获取函数地址的方式进行判断。具体的操作请参考有关动态链接库编程的章节。下面代码演示了如何获取磁盘空间信息。

```
01  void CDiskInfoDlg::OnButtonGetfreespace()//获取磁盘空间信息
02  {
03      UpdateData(true);             //从控件中更新数据，更新要获取的磁盘名称
04      DWORD   lSPC,lBPS,lNOFC,lTNOC;  //存放磁盘空间的变量
05      //获取磁盘空间信息
06      if (GetDiskFreeSpace(m_DiskName,&lSPC, &lBPS, &lNOFC, &lTNOC))
07      WriteLog("%s 盘的磁盘空间情况：\r\n 空闲字节数=%d;每柱面包含的扇区数=%d; \
08              \r\n 每扇区包含的字节数=%d;空闲柱面数=%d;总柱面数=%d.",
09              m_DiskName, lSPC*lBPS*lNOFC, lSPC, lBPS, lNOFC, lTNOC);
10      //显示获取的磁盘空间信息
11      else
12          WriteLog("获取磁盘空间信息失败");//显示错误提示
13  }
```

上面代码调用 GetDiskFreeSpace()函数获取指定驱动器的空闲磁盘信息，并根据返回值计算总的空闲磁盘空间。程序运行效果如图 20-7 所示。

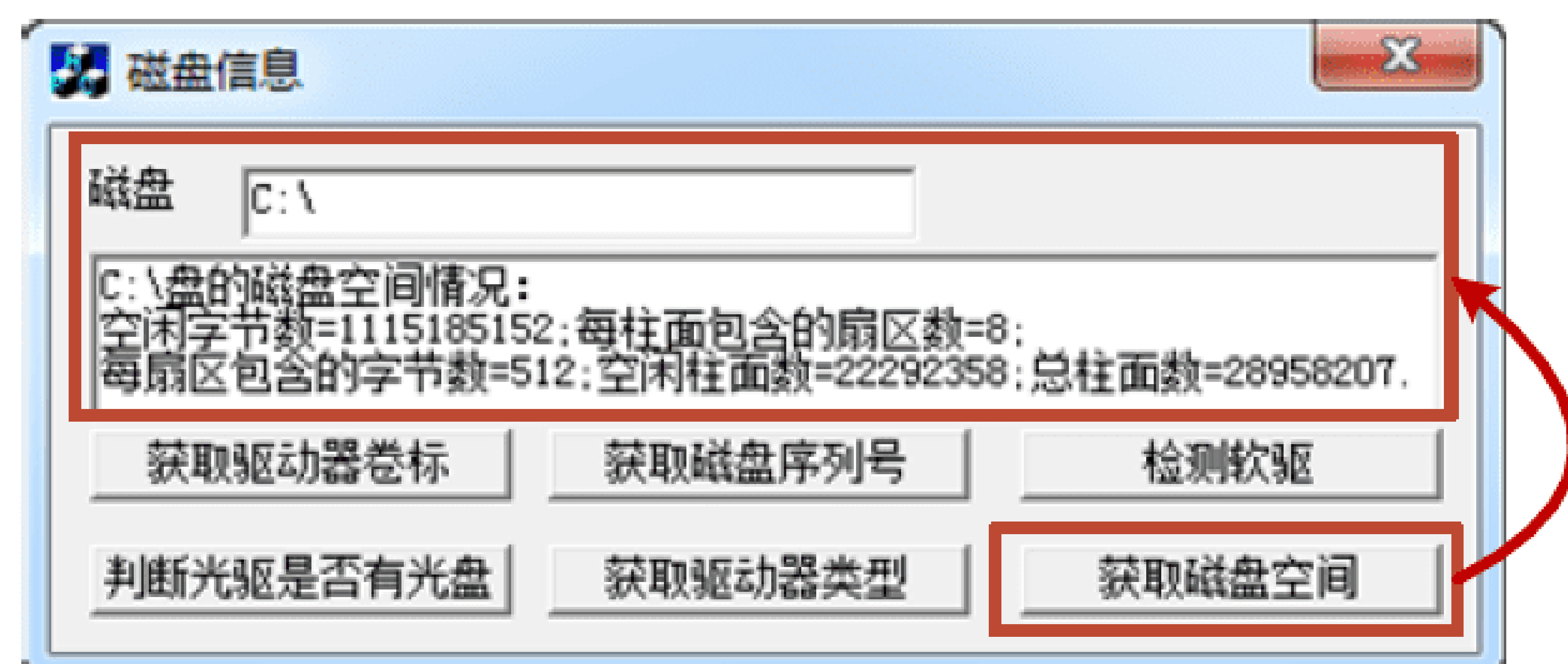


图 20-7　获取磁盘空间信息运行效果

20.2　操 作 磁 盘

在 VC 中，可以通过调用 Win32 API 执行有关磁盘的操作，比如格式化磁盘、控制磁盘共享状态、设置磁盘卷标、进行磁盘碎片整理、转换磁盘格式、显示和隐藏磁盘分区、

更改磁盘分区号以及监视硬盘操作等。本节以实例介绍如何实现这些磁盘操作。

20.2.1　格式化磁盘

在 Windows 平台中，可以调用 Win32 API 函数 SHFormatDrive()格式化磁盘。该函数在 Shell32.dll 动态库中实现，因此，需要动态装载 Shell32 动态库，并获取 SHFormatDrive()函数的地址进行操作。代码如下：

```
DWORD SHFormatDrive(
    HWND hwnd,              //进行格式化的窗口
    UINT drive,             //要格式化的驱动器
    UINT fmtID,             //默认格式化，SHFMT_ID_DEFAULT
    UINT options            //选项，完全格式化 SHFMT_OPT_FULL 和系统格式化
);                          //SHFMT_OPT_SYSONLY
```

上面代码说明了格式化磁盘 SHFormatDrive()函数的函数原型。SHFMT_ERROR、SHFMT_CANCEL 和 SHFMT_NOFORMAT 宏，分别表示格式化发生错误、取消格式化和不能对相应磁盘进行格式化。下面代码显示了如何动态装载 Shell32.dll，并调用 SHFormatDrive()函数格式化磁盘函数。

```
01  void CDiskOperDlg::OnButtonFormat()//格式化磁盘
02  {
03      UpdateData(false);              //更新控件数据变量的取值，从控件获取值
04      UINT uiDriver = m_DiskName[0] - 'A';//计算要格式化的磁盘的号码
05      HINSTANCE  hInstance=LoadLibrary(_T("Shell32.dll"));
06      //装载 Shell32.dll 动态库
07      if (hInstance == NULL)
08          return;                     //装载失败，则返回
09      //获取 SHFormatDrive()函数的指针
10      PFUNCTIONFORMAT pFunctionFormat=(PFUNCTIONFORMAT)GetProcAddress
11          (hInstance, _T("SHFormatDrive"));
12       //如果 SHFormatDrive()函数指针为 NULL
13      if(pFunctionFormat==NULL)
14      {
15          FreeLibrary(hInstance);     //释放对动态库的实例引用
16          return;                     //返回
17      }
18      DWORD dwResult = (pFunctionFormat)(this->m_hWnd, uiDriver,
19          SHFMT_ID_DEFAULT, SHFMT_OPT_FULL);       //格式化磁盘
20      switch (dwResult)                            //判断格式化磁盘的返回值
21      {
22      case SHFMT_ERROR:                            //格式化磁盘发生错误
23          WriteLog("格式化磁盘%s 发生错误" , m_DiskName);
24          break;
25      case SHFMT_CANCEL:                           //取消格式化磁盘
26          WriteLog("取消格式化磁盘%s " , m_DiskName);
27          break;
28      case SHFMT_NOFORMAT:                         //不能格式化磁盘
29          WriteLog("不能格式化磁盘%s " , m_DiskName);
30          break;
31      default:                                     //格式化磁盘成功
32          WriteLog("格式化磁盘%s 成功" , m_DiskName);
33          break;
```

```
34        }
35        FreeLibrary(hInstance);                          //释放对动态库的实例引用
36        return;                                          //函数返回
37    }
```

上面代码首先调用了 LoadLibrary()函数装载动态链接库，然后调用 GetProcAddress()函数获取 SHFormatDrive()函数的地址，并调用此函数格式化文本框中输入的驱动器，此处使用驱动器名称与 A 之间的差值，作为第二个参数，即相对于 A 盘来说是第几个驱动器，最后判断格式化的返回值，并调用 FreeLibrary()函数释放对动态链接库的调用。调用此函数后，会调用系统中用于格式化磁盘的对话框。如果用户单击“取消”按钮，则返回值会表示格式化磁盘失败，如果成功格式化，则程序会提示格式化磁盘成功。如图 20-8 所示为单击“磁盘格式化”按钮后的弹出界面。成功格式化 G 盘后，程序的运行效果如图 20-9 所示。

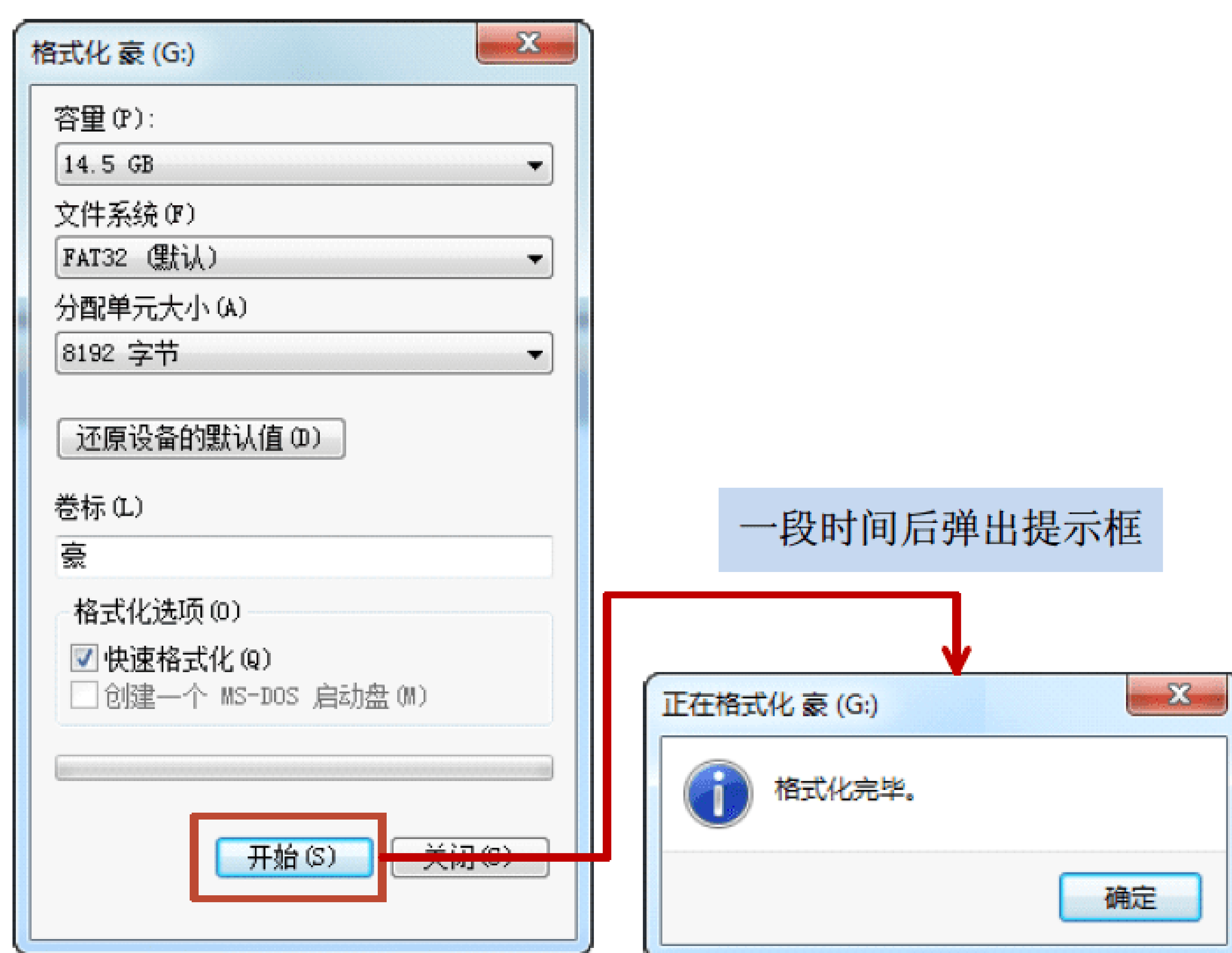


图 20-8　格式化磁盘弹出对话框

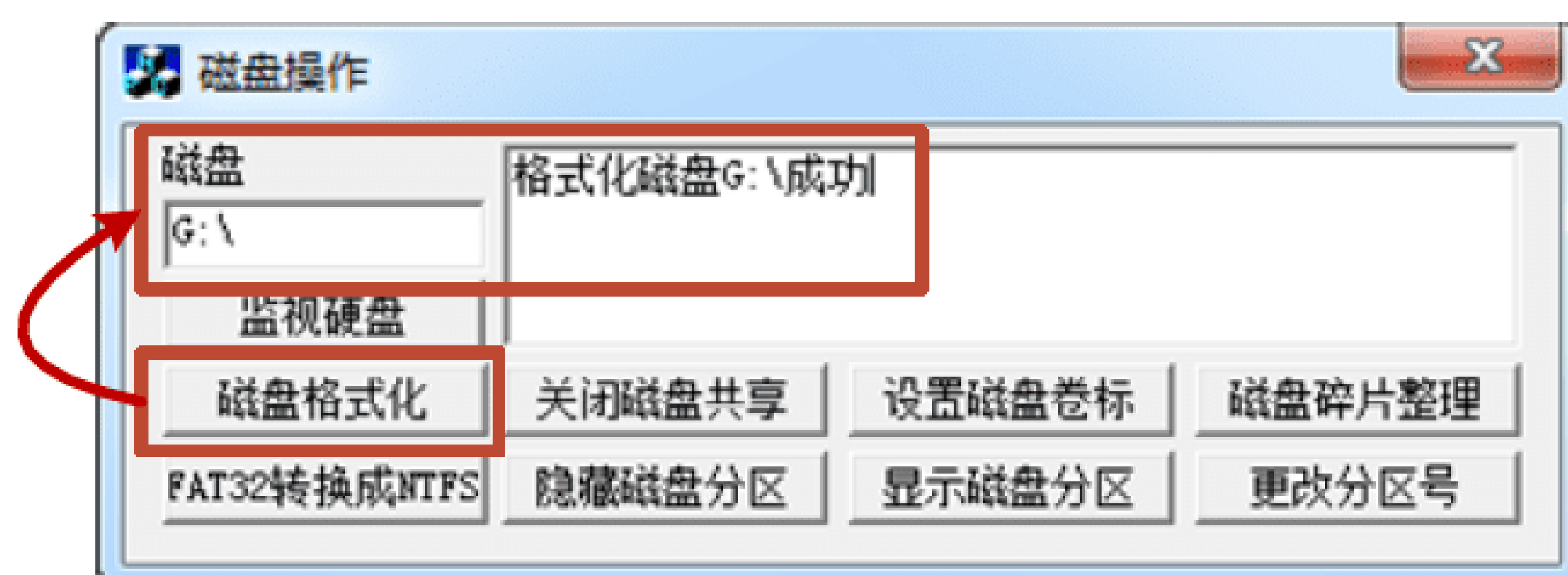


图 20-9　成功格式化磁盘后的程序运行效果

20.2.2　关闭磁盘共享

调用 NetShareDel()函数可以删除指定计算机上的共享资源，并断开与共享资源的所有连接。只有 Administrator 用户和管理员本地组用户才可以执行此函数。其函数原型为：

```
NET API STATUS NetShareDel(
    LPWSTR servername,       //要关闭的共享资源所在的计算机名称
    LPWSTR netname,          //要删除的共享资源的网络名称 Unicode 字符串
    DWORD reserved );        //预留，必须为 0
```

如下代码为实现关闭指定磁盘共享的代码。

```
01  void CDiskOperDlg::OnButtonDelshare()       //关闭磁盘共享
02  {
03      UpdateData(true);                       //从控件获取数据
04      WCHAR wsz[MAX PATH];                    //定义 Unicode 字符数组
05      CString disk;                           //定义磁盘共享名称变量
06      disk.Format("%c$", m DiskName[0]);      //格式化磁盘共享名称
07      //将共享名称转换到 Unicode 字符数组
08      wcscpy(wsz, disk.AllocSysString());
09      NET API STATUS dwStatus = NetShareDel(NULL,(LPWSTR)wsz,0);
10      //关闭磁盘共享
11      switch(dwStatus)                    //判断操作返回值，并输出提示信息
12      {
13      case NERR Success:
14          WriteLog("成功关闭%s 磁盘共享\n", disk);
15          break;
16      case ERROR ACCESS DENIED:
17          WriteLog("关闭%s 磁盘共享错误\n 原因=用户没有权限执行此项操作。",disk);
18          break;
19      case ERROR INVALID PARAMETER:
20          WriteLog("关闭%s 磁盘共享错误\n 原因=参数无效。", disk);
21          break;
22      case ERROR NOT ENOUGH MEMORY:
23          WriteLog("关闭%s 磁盘共享错误\n 原因=内存不足。", disk);
24          break;
25      case NERR NetNameNotFound:
26          WriteLog("关闭%s 磁盘共享错误\n 原因=不存在此共享名称。",disk);
27          break;
28      default:
29          WriteLog("关闭%s 磁盘共享错误\n 原因=未知。",disk);
30          break;
31      }
32  }
```

上面代码调用 NetShareDel()函数删除本地计算机上的指定磁盘共享。因为是本地计算机，所以函数的第一个参数为 NULL，第二个参数使用 X$表示要关闭 X 盘的共享。需要注意的是，共享资源的名称必须是 Unicode 字符串，否则函数会执行失败。运行此函数后，本地计算机上的 X 盘共享就关闭了。程序运行效果如图 20-10 所示。

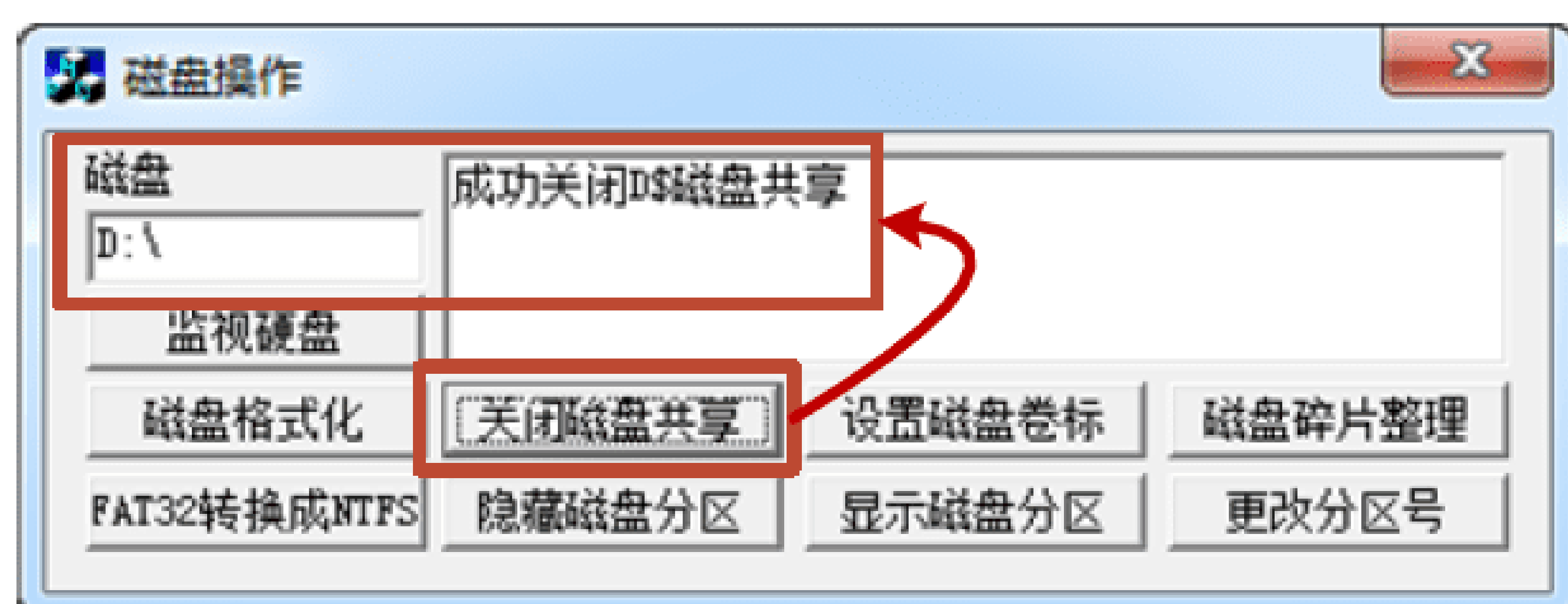


图 20-10　关闭磁盘共享运行效果

20.2.3　设置磁盘卷标

调用 Win32 API 函数 SetVolumeLabel()可以设置指定驱动器的磁盘卷标。其函数原型为：

```
BOOL SetVolumeLabel(
  LPCTSTR lpRootPathName,       //指向要设置卷标的磁盘根目录,默认为当前目录
  LPCTSTR lpVolumeName );       //指向要设置的卷标名称,默认为当前卷标
```

以下代码演示了如何设置磁盘卷标。

```
01  void CDiskOperDlg::OnButtonSetvolum()//设置磁盘卷标测试函数
02  {
03      UpdateData(true);                      //从控件获取数据
04      //设置磁盘的卷标为“我的系统盘”
05      if (SetVolumeLabel(m_DiskName, "我的系统盘"))
06          WriteLog("设置磁盘卷标成功");   //如果设置成功，则显示成功提示信息
07      else
08          WriteLog("设置磁盘卷标失败");   //如果设置失败，则显示失败提示信息
09  }
```

上面代码调用 SetVolumeLabel()函数设置文本框中指定的磁盘卷标为“我的系统盘”，并在日志文本框中显示操作结果。程序运行效果如图 20-11 所示。

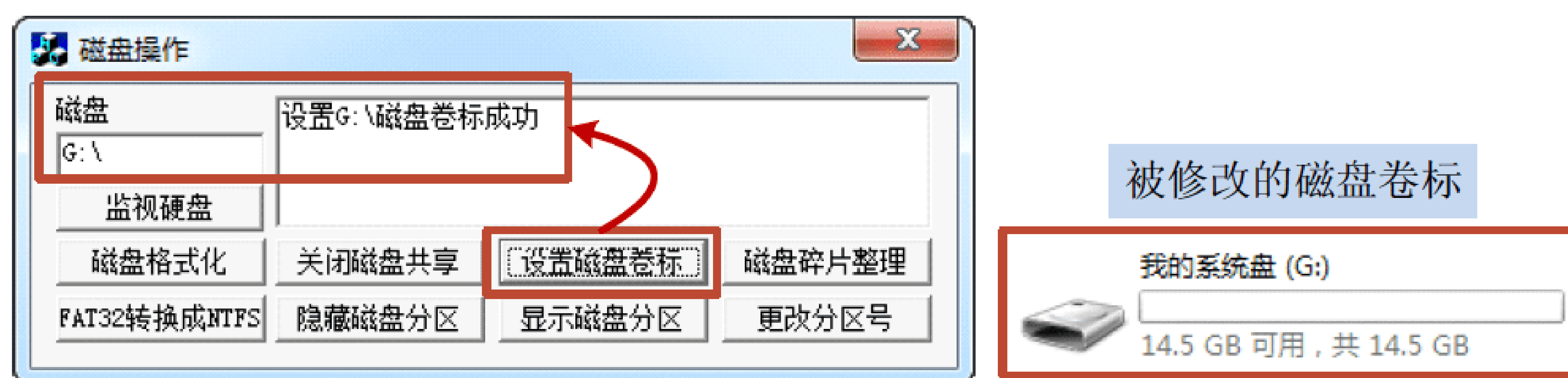


图 20-11　设置磁盘卷标运行效果

20.2.4　磁盘碎片整理

Windows 的命令行工具 defrag 可以对磁盘进行碎片整理。在程序中，使用 system()函数或 WinExec()函数调用此命令，可以执行磁盘碎片整理。其语法格式为：

```
defrag [drive: | /all] [/F | /U | /Q] [/noprompt] [/concise | /detailed]
```

其中各参数的含义如下。

- ❑ drive 参数是要进行磁盘碎片整理的磁盘的驱动符。
- ❑ /all 参数会对所有本地的不可移除的磁盘进行磁盘碎片整理。
- ❑ /F 参数表示整理文件并释放空间。
- ❑ /U 参数表示仅整理文件。
- ❑ /Q 参数表示仅整理空闲空间。
- ❑ /concise 表示隐藏详细视图。

- ❑ /detailed 表示显示详细视图。
- ❑ /noprompt 表示在磁盘碎片整理过程中不显示提示信息。

具体调用方法如下：

```
01  void CDiskOperDlg::OnButtonSpzl()//实现磁盘碎片整理的处理函数
02  {
03      system("defrag E:");      //调用 defrag 命令行工具进行 E 盘的磁盘碎片整理
04  }
```

上面的代码会对系统中的 E 盘进行碎片整理。运行效果会快速出现然后消失，如图 20-12 所示为在命令行下的模拟效果。

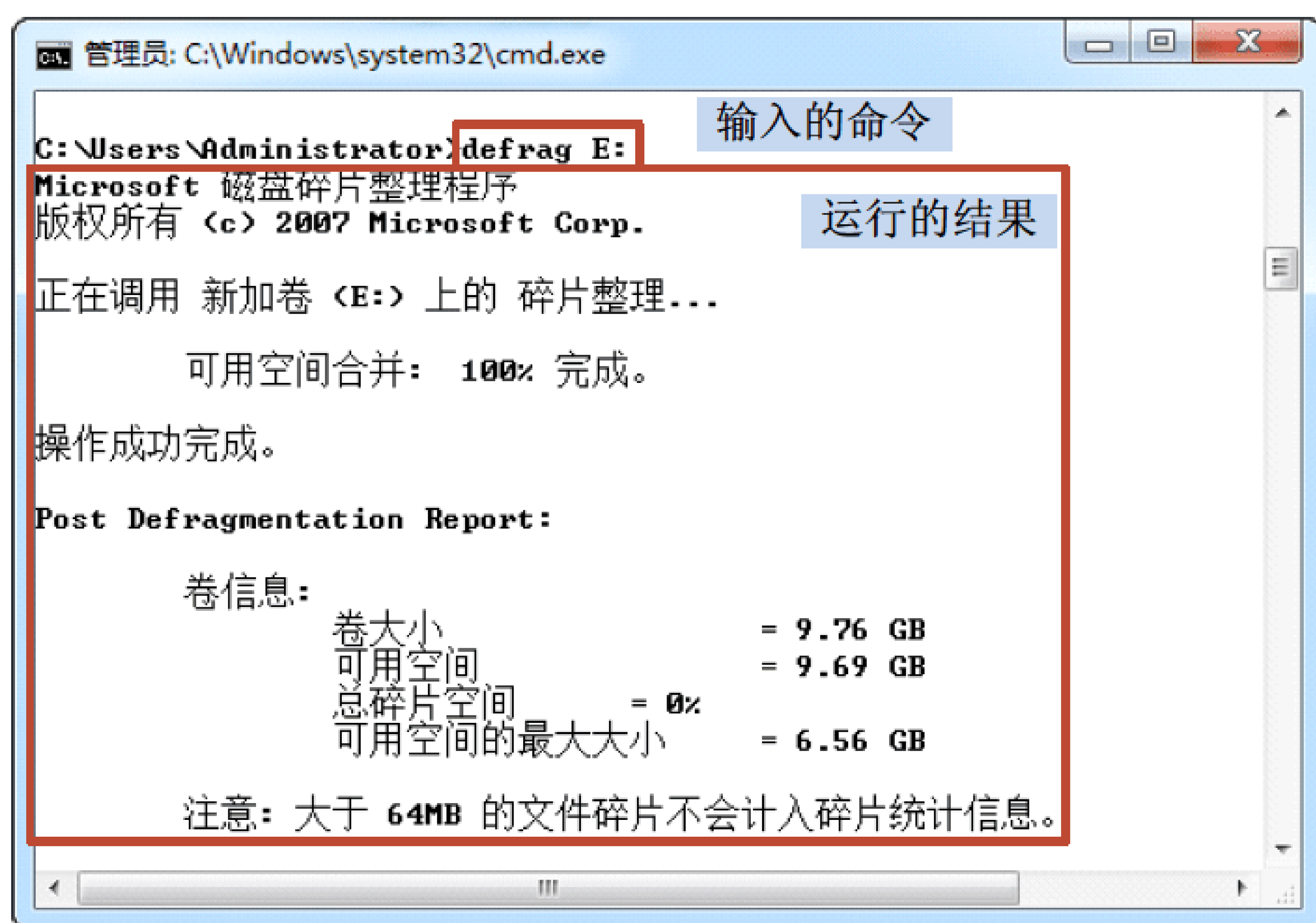


图 20-12　磁盘碎片整理运行效果

20.2.5　从 FAT32 转换为 NTFS

DOS 命令 convert 可以将磁盘从 FAT32 转换成 NTFS 格式。程序中可以通过调用此 DOS 命令，将指定磁盘从 FAT32 格式转换成 NTFS 格式。其语法格式为：

```
convert [Volume] /fs:ntfs [/v] [/cvtarea:FileName] [/nosecurity] [/x]
```

其中，Volume 参数用于指定驱动器号，后面跟一个冒号，表示装入点或要转换为 NTFS 的卷名。/fs:ntfs 表示将卷转换为 NTFS。/v 表示在转换期间显示所有的消息。在程序中调用此 DOS 命令的代码如下：

```
01  //从 FAT32 转换为 NTFS 格式的处理函数
02  void CDiskOperDlg::OnButtonFat32tontfs()
03  {
04      system("convert E: /FS:NTFS");
05                              //调用 convert 命令行工具，从 FAT32 转换为 NTFS
06      system("pause");        //等待工具运行
07  }
```

上面代码会将系统中的 E 盘，从 FAT32 转换为 NTFS 格式，读者在测试示例时要注意数据备份。程序运行的效果如图 20-13 所示。

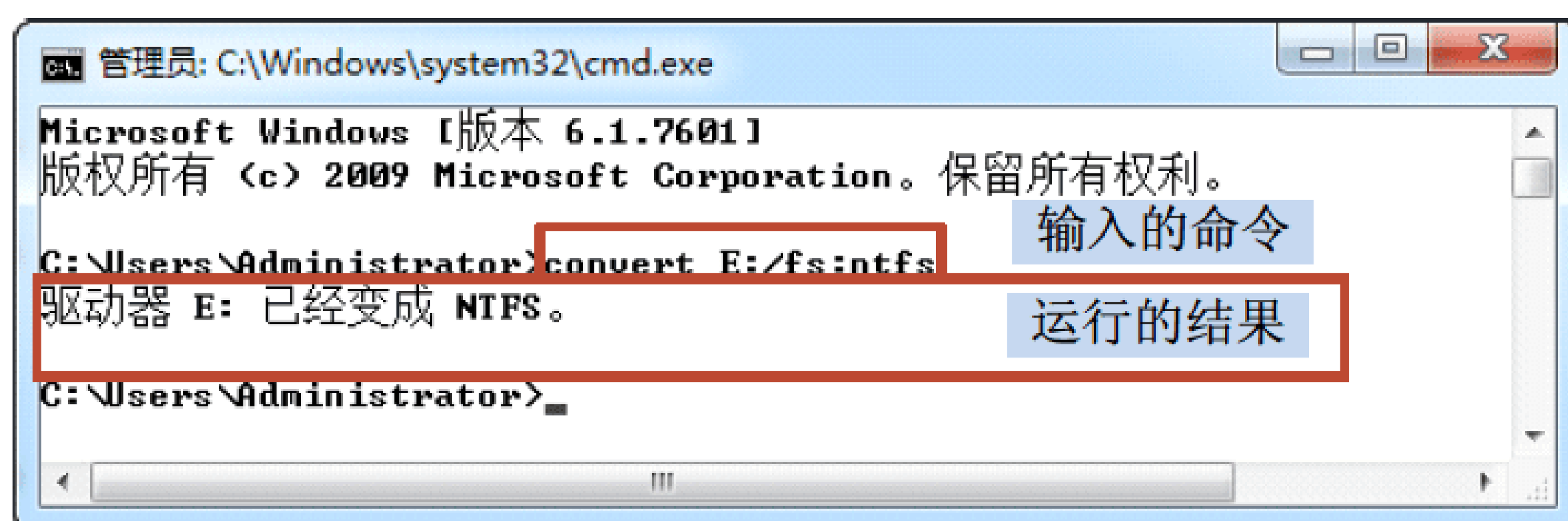


图 20-13　从 FAT32 转换为 NTFS 运行效果

20.2.6　隐藏磁盘分区

在 Windows 中调用 QueryDosDevice()函数可以通过设备名称获取 MS-DOS 设备对应路径的信息，DefineDosDevice()函数可以定义、重命名或删除指定 MS-DOS 设备名称。其函数原型为：

```
DWORD QueryDosDevice(          //查询指定分区对应的设备路径
    LPCTSTR lpDeviceName,      //要查询路径的 MS-DOS 设备的名称
    LPTSTR lpTargetPath,       //存放查询结果的缓冲区
    DWORD ucchMax );           //指定 lpTargetPath 缓冲区的大小
BOOL DefineDosDevice(          //定义 MS-DOS 设备名称
    DWORD dwFlags,             //指定函数执行的操作
    LPCTSTR lpDeviceName,      //要进行设置的 MS-DOS 设备名称
    LPCTSTR lpTargetPath);     //要进行设置的 Windows 路径名称
```

其中，dwFlags 参数用于指定函数执行的操作，有效取值有 DDD_RAW_TARGET_PATH、DDD_REMOVE_DEFINITION 和 DDD_EXACT_MATCH_ON_REMOVE，分别用于表示为 MS-DOS 设备定义 Windows 路径、删除 MS-DOS 设备对应的路径信息以及匹配删除路径定义。以下代码演示了如何隐藏磁盘分区。

```
01  void CDiskOperDlg::OnButtonHidedisk()                //隐藏磁盘分区
02  {
03      UpdateData(true);                                //从控件获取数据
04      memset(szPath,0x00, sizeof(szPath));             //初始化路径变量
05      CString csDisk;                                  //定义磁盘名称
06      csDisk.Format("%c:", m_DiskName[0]);             //格式化要隐藏的磁盘名称
07      //查询磁盘对应的设备路径
08      if (QueryDosDeviceA(csDisk, szPath, MAX_PATH) == 0)
09      {
10          WriteLog("获取磁盘分区%s 路径名失败", csDisk);//查询失败，显示结果
11          return;                                      //查询失败，返回
12      }
13      //隐藏磁盘分区
14      if (DefineDosDeviceA(DDD_REMOVE_DEFINITION, csDisk, NULL))
15          WriteLog("隐藏磁盘分区%s 成功。\r\n 路径名称=%s",csDisk, szPath);
16      else
17          WriteLog("隐藏磁盘分区%s 失败。\r\n 路径名称=%s",csDisk, szPath);
18  }
```

上面代码，首先调用 QueryDosDeviceA()函数获取指定分区的路径。然后调用 DefineDosDeviceA()函数，并传入 DDD_ REMOVE_DEFINITION 参数删除设备路径与分区号的对应关系，从而隐藏磁盘分区，最后将结果显示在日志文本框中。程序运行后，指定的磁盘分区就被隐藏了，运行结果如图 20-14 所示。

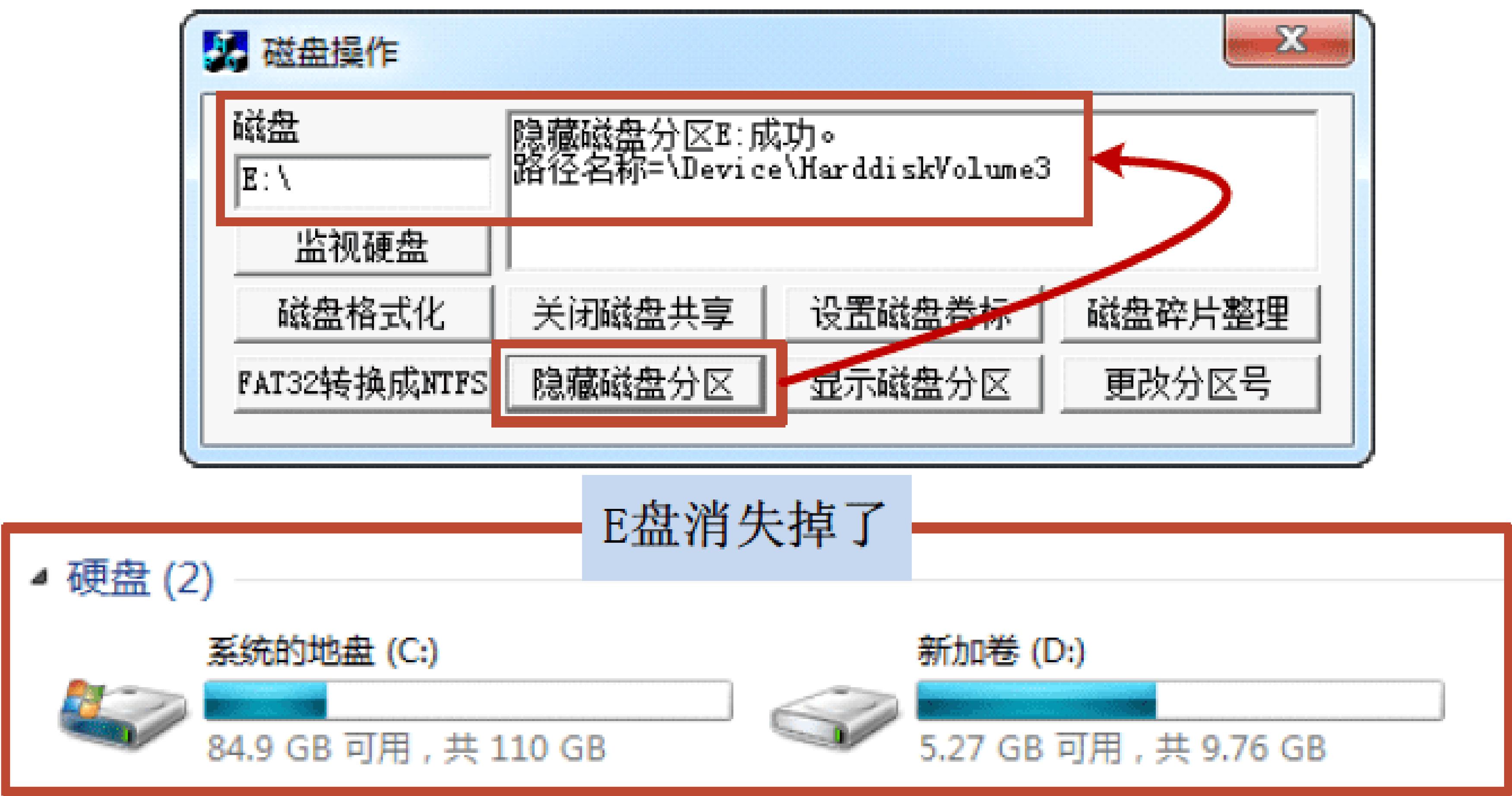


图 20-14　隐藏磁盘分区运行效果

20.2.7　显示被隐藏的磁盘分区

与 21.2.6 小节介绍的方法相同，使用前面保存下来的路径信息和设备名称，可以显示被隐藏的磁盘分区。具体代码如下：

```
01  void CDiskOperDlg::OnButtonShowdisk()       //显示被隐藏的磁盘分区
02  {
03      UpdateData(true);                       //从控件获取数据
04      CString csDisk;                         //定义磁盘名称
05      csDisk.Format("%c:", m_DiskName[0]);    //格式化要隐藏的磁盘名称
06      //增加指定设备的磁盘名称
07      if (DefineDosDevice(DDD_RAW_TARGET_PATH, csDisk, szPath))
08          WriteLog("显示被隐藏的磁盘分区%s 成功。\r\n 路径名称=%s",
09                      csDisk,szPath);
10      else
11          WriteLog("显示被隐藏的磁盘分区%s 失败。\r\n 路径名称=%s",
12                      csDisk,szPath);
13  }
```

上面代码中，调用 DefineDosDevice()函数传入 DDD_RAW_TARGET_PATH 参数、分区名称和通过 QueryDosDevice()函数返回的路径信息，重新显示刚才被隐藏的磁盘分区。程序运行后，刚才被隐藏的磁盘分区会重新显示出来。运行效果如图 20-15 所示。

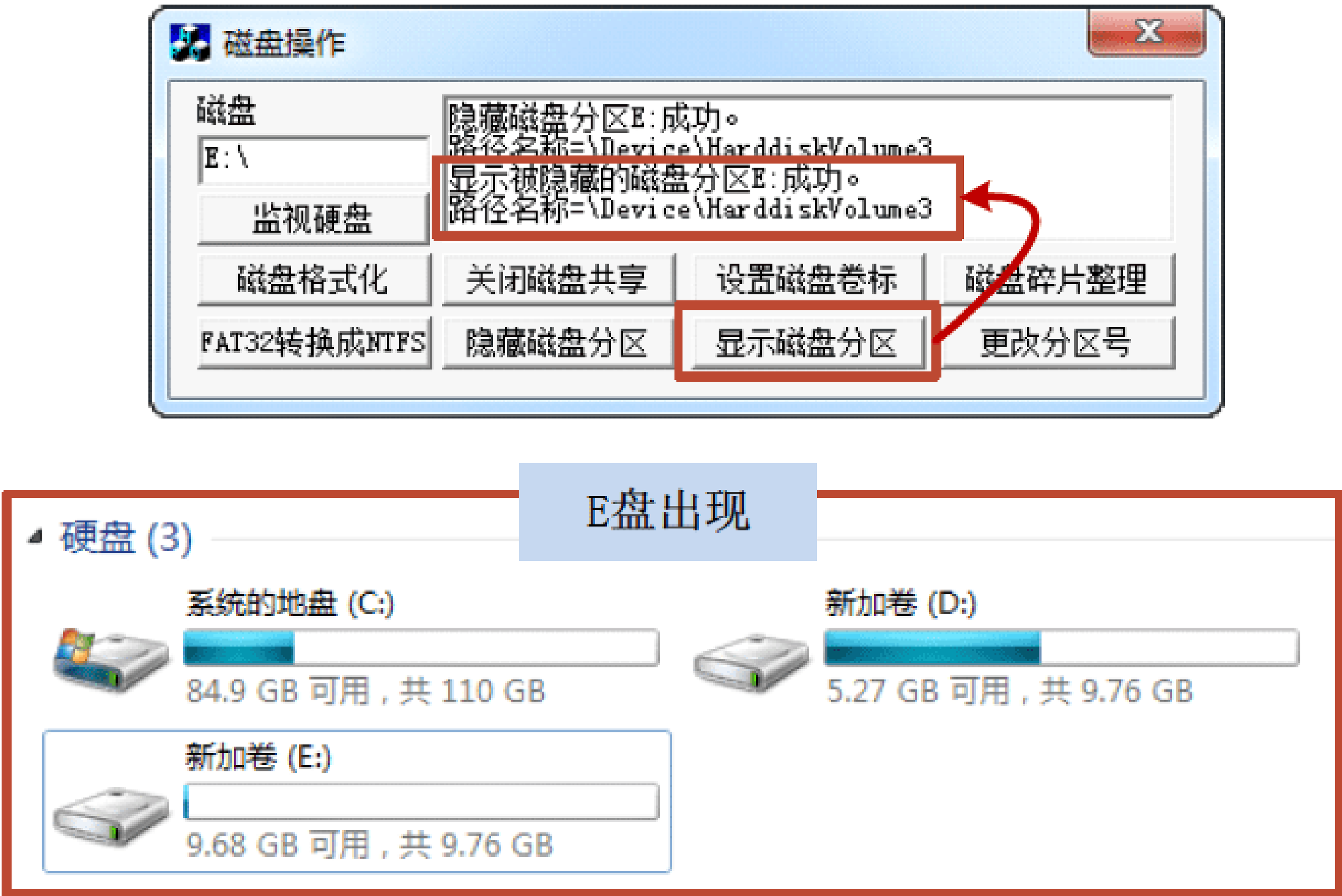


图 20-15　显示被隐藏的磁盘分区运行效果

20.2.8　如何更改分区号

通过调用 Windows 内核 API，可以实现磁盘管理器中的更改分区号的功能。首先需要调用 GetVolumeNameForVolumeMountPointA()函数获取指定分区号的分区卷标，此函数有 3 个参数，分别是要获取卷标的分区号的字符串、存放获取的卷标的字符串和返回的卷标所占用的空间。通过传入分区号调用 DeleteVolumeMountPointA()函数删除指定的分区号。调用 SetVolumeMountPointA()函数为指定卷标分配新的分区号，此函数有两个参数，表示要设置的分区号和要设置分区号的分区的卷名。下面的代码演示了使用这 3 个函数如何更改分区号。

```
01  //定义 GetVolumeNameForVolumeMountPointA 函数原型
02  typedef BOOL  (WINAPI *PROCGET)  (LPCTSTR,LPCTSTR,LPDWORD);
03  typedef BOOL  (WINAPI *PROCDEL)  (LPCTSTR);
04  //定义 DeleteVolumeMountPointA 函数原型
05  typedef BOOL  (WINAPI *PROCSET)  (LPCTSTR,LPCTSTR);
06  //定义 SetVolumeMountPointA 函数原型
07  void CDiskOperDlg::OnButtonUpdatediskno()  //更改分区号
08  {
09      UpdateData(true);                      //从控件获取数据
10      HMODULE hKernel = GetModuleHandle("kernel32");
11      //装载 kernel32.dll
12      if (hKernel)                           //如果装载成功，则继续
13      {
14          PROCGET getAPI=(PROCGET)GetProcAddress(hKernel,
15              "GetVolumeNameForVolumeMountPointA");    //获取函数入口
16          //获取 DeleteVolumeMountPointA()函数入口
17          PROCDEL delAPI = (PROCDEL)GetProcAddress(hKernel,
18                                  "DeleteVolumeMountPointA");
19              //获取 SetVolumeMountPointA()函数入口
20              PROCSET setAPI = (PROCSET)
21                 GetProcAddress(hKernel,"SetVolumeMountPointA");
22              char szVolName [MAX_PATH];               //定义卷名变量
23          DWORD dwLen;                                 //定义卷名长度变量
24          if (!getAPI(m_DiskName, szVolName, &dwLen)) //查询指定分区的卷名
25          {
26              WriteLog("查询分区%s 的卷名失败。", m_DiskName);
27              return;                                  //查询失败，返回
28          }
29          if (!delAPI(m_DiskName))                //删除卷名对应的分区号
30          {
31              WriteLog("删除卷名为%s 对应的%s 分区号。",
32                          szVolName, m DiskName);
33              return;                                  //删除失败，返回
34          }
35          if (!setAPI(_T("Z:\\"), szVolName))      //为卷设置新的分区号为 Z
36          {
37              WriteLog("将卷名为%s 的分区号改为 Z:失败。", szVolName);
38              return;                                  //设置失败，返回
39          }
40          WriteLog("将卷名为%s 的分区号改为 Z:成功。\r\n 原来的分区号=%s",
41              szVolName, m_DiskName);
```

```
42      }
43      else
44          WriteLog("装载 kernel32 失败！");              //显示装载 DLL 失败信息
45  }
```

上面代码按照前面所讲的方法，分 3 步更改分区号。因为这 3 个函数不是公开的 API 函数，因此，需要使用 typedef 定义函数原型，并使用 GetProcAddress()函数从 DLL 库中动态加载，获取函数指针，通过函数指针调用这 3 个函数。程序的运行效果如图 20-16 所示。

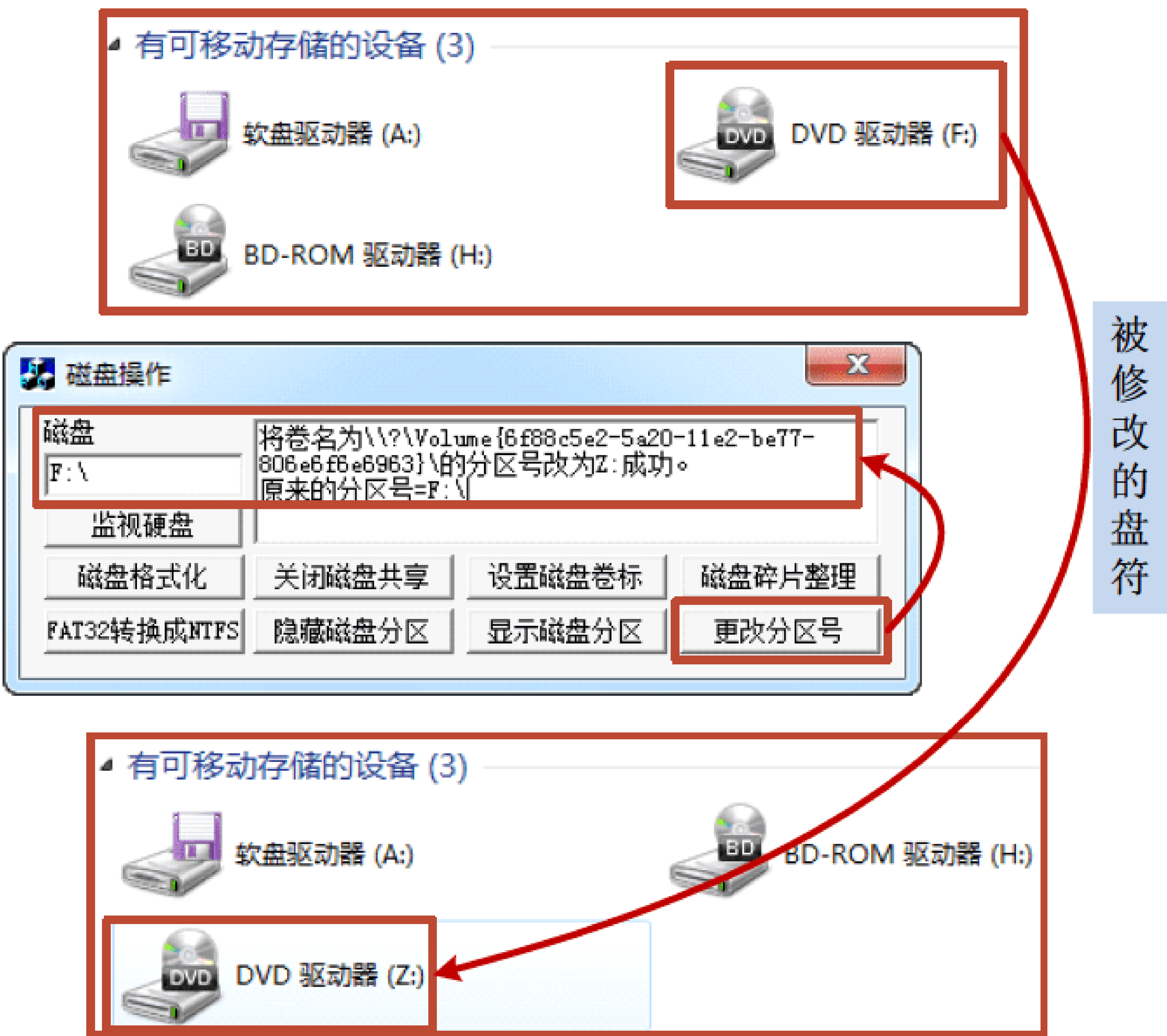


图 20-16　更改分区号的运行效果

20.2.9　如何监视硬盘

调用 FindFirstChangeNotification()函数和 FindNextChangeNotification()函数可以完成对磁盘操作的监视。其中 FindFirstChangeNotification()函数，确定开始监视的参数，返回值为监视句柄。将该返回值传入 FindNextChangeNotification()函数可以继续进行对磁盘的监视，使用 WaitForSingleObject()函数等待事件的发生。当监视到一次动作时，可以继续调用 FindNextChangeNotification()函数继续等待下一次对磁盘的操作。其中，可以监视的磁盘操作如表 20-3 所示。

表 20-3　可以监视的磁盘操作

操　作　值	操 作 含 义
FILE_NOTIFY_CHANGE_FILE_NAME	监视指定目录中的文件名称改变的操作
FILE_NOTIFY_CHANGE_DIR_NAME	监视指定目录中的目录名称改变的操作
FILE_NOTIFY_CHANGE_ATTRIBUTES	监视指定目录中的文件或目录属性改变的操作
FILE_NOTIFY_CHANGE_SIZE	监视指定目录中的文件写操作
FILE_NOTIFY_CHANGE_LAST_WRITE	监视指定目录中的最近一次文件写入磁盘的操作
FILE_NOTIFY_CHANGE_SECURITY	监视指定目录中的文件或目录的安全属性的修改操作

下面显示了监视磁盘操作的代码。

```
01  void CDiskOperDlg::OnButtonMonitorDisk()    //监视磁盘操作实现函数
02  {
03      DWORD dwWaitStatus;                     //定义等待状态值
04      HANDLE dwChangeHandle;                  //定义修改句柄
05      dwChangeHandle=FindFirstChangeNotification(m_DiskName, true,
06          FILE_NOTIFY_CHANGE_FILE_NAME);//获取磁盘上第一个修改文件名的通知句柄
07      if (dwChangeHandle==INVALID_HANDLE_VALUE)
08          return;                             //如果句柄获取无效，则返回
09      bMonitor = true;                        //定义继续循环变量为 true
10      WriteLog("正在监视%c 盘重命名操作......", m_DiskName[0]);
11      while (bMonitor)                        //循环监视磁盘操作
12      {
13          dwWaitStatus = WaitForSingleObject(dwChangeHandle, INFINITE);
14          //等待修改磁盘操作
15          if (dwWaitStatus == WAIT_OBJECT_0)  //如果检测到有事件发生
16          {
17              MessageBox(m_DiskName, "检测到文件重命名操作");//显示提示信息
18              //如果启动下一次重命名文件监控失败，则退出循环
19              if (FindNextChangeNotification(dwChangeHandle) == false)
20                  bMonitor = false;
21          }
22          Sleep(100);                         //程序暂停 0.1 秒
23      }
24  }
```

上面代码首先调用了 FindFirstChangeNotification()函数启动对指定盘的操作监视，将返回句柄值 dwChangeHandle 传入 FindNextChangeNotification()函数继续等待。其中，调用 WaitForSingleObject()函数获取发生的事件。此过程的处理应该启动线程专门处理监视过程，此处为了简单地说明功能的实现，在主进程中演示。程序的运行效果如图 20-17 所示。

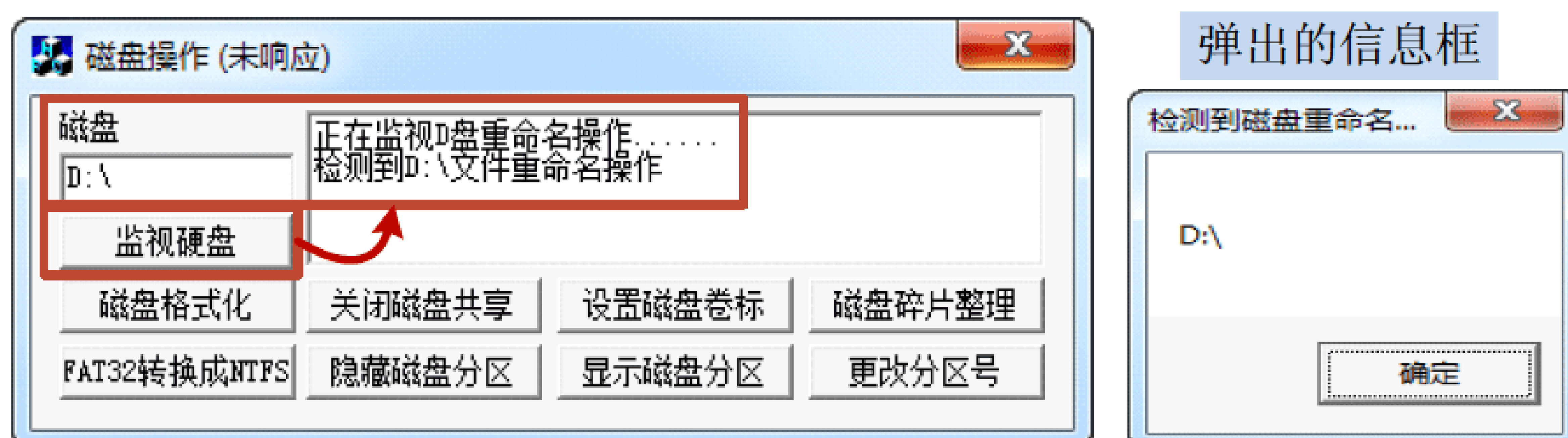


图 20-17　监视磁盘操作的运行效果

20.3　系统控制与调用

Windows 操作系统 SDK 提供一组函数和消息，可以实现对系统的控制和调用。包括从程序中调用外部程序、调用创建快捷方式的向导、访问控制面板中的各项、控制光驱的弹开和关闭、控制操作系统的关机、控制显示器的电源状态、控制屏幕保护程序的状态、控制输入法的开关、播放系统提示音以及列举程序中的可执行文件等。本节将介绍如何实现这些功能。

20.3.1　调用外部程序

在程序中，调用 Windows API 函数 CreateProcess()可以启动执行指定可执行文件的新进程，并且可以在其中设置执行参数。其函数原型为：

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,    //指向要运行的可执行模块
  //指向命令行字符串的指针，为 NULL 时使用模块名称指定运行的命令行
  LPTSTR lpCommandLine,
  LPSECURITY ATTRIBUTES lpProcessAttributes,   //进程安全属性
  LPSECURITY_ATTRIBUTES lpThreadAttributes,    //线程安全属性
  BOOL bInheritHandles,         //指定新进程是否可以从调用的进程中继承句柄
  DWORD dwCreationFlags,        //指定控制优先级类和进程创建的附加标记
  LPVOID lpEnvironment,         //指向新环境块
  LPCTSTR lpCurrentDirectory,   //指向新进程的环境块
  //指向 STARTUPINFO 结构指定新进程主对话框如何显示的结构
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation);//指向用于接收新进程信息的结构
```

以下代码是在程序中运行记事本的实现。

```
01  void CSysControlSampleDlg::OnButtonExe()//调用外部程序的实现函数
02  {
03      STARTUPINFO    si;              //定义存放启动信息的 STARTUPINFO 结构变量
04      PROCESS_INFORMATION    pi;      //定义存放进程信息的 PROCESS_INFORMATION
05                                      //结构变量
06      HANDLE  hProcess,hThread;       //分别定义进程句柄和线程句柄
07      si.cb =sizeof(STARTUPINFO);     //初始化 si 变量的长度为 STARTUPINFO 结构的
08                                      //长度
09      si.lpReserved = NULL;           //初始化 si 变量的 lpReserved 成员为 NULL
10      si.lpDesktop=NULL;              //初始化 si 变量的 lpDesktop 成员为 NULL
11      si.lpTitle=NULL;                //初始化 si 变量的标题 lpTitle 成员
12      si.cbReserved2=0;               //初始化 si 变量的 cbReserved2 成员为 0
13      si.lpReserved2=NULL;            //初始化 si 变量的 lpReserved2 成员为 NULL
14      BOOL bResult =  CreateProcess("C:\\WINDOWS\\NOTEPAD.EXE", NULL,
15          NULL, NULL, true, 0, NULL,NULL, &si,&pi);//创建打开记事本的进程
16      if (bResult)                    //如果返回的结果为 true
17      {
18          hProcess = pi.hProcess; //保存进程句柄到 hProcess 变量中
19          hThread = pi.hThread;     //保存进程句柄到 hThread 变量中
20          WriteLog("调用外部程序成功。\n 进程 Id = %d\n 线程 Id = %d\n",
21              pi.dwProcessId,pi.dwThreadId);//显示日志信息
22      }
23      else
24          //显示错误信息提示
25          WriteLog ("调用记事本失败。错误代码=%X",GetLastError());
26  }
```

上面代码定义了启动信息变量 si 和进程信息变量 pi，为 si 初始化取值后，调用 CreateProcess()函数运行记事本程序，并保存进程句柄输出信息提示。程序运行效果如图 20-18 所示。

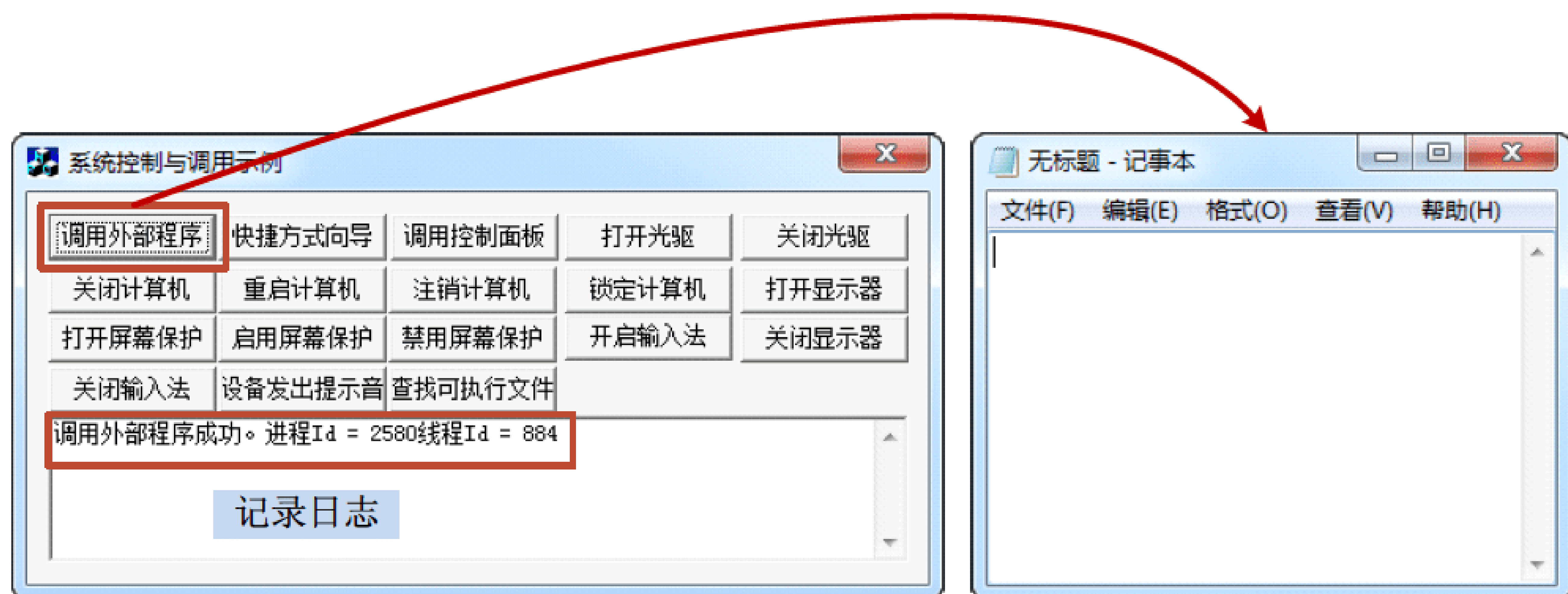


图 20-18　调用外部程序运行效果

20.3.2　调用创建快捷方式向导

调用 WinExec()函数可以运行指定的应用程序，因此使用这个函数就可以调用创建快捷方式的向导程序。其函数原型为：

```
UINT WinExec(
  LPCSTR lpCmdLine,              //表示要运行的命令
  UINT uCmdShow);                //表示是否显示程序界面
```

创建快捷方式向导在 rundll32.exe 中的 appwiz.cpl 中实现，代码如下：

```
01  WinExec("rundll32.exe appwiz.cpl,NewLinkHere C:\\", SW_SHOW);
```

NewLinkHere 空格后的 C:\\表示创建快捷方式存放的路径，读者可以根据自己的需要设置。运行此代码，则会弹出创建快捷方式向导，使用此向导创建的快捷方式 lnk 文件会存放在 C:\目录下。程序运行效果如图 20-19 所示。

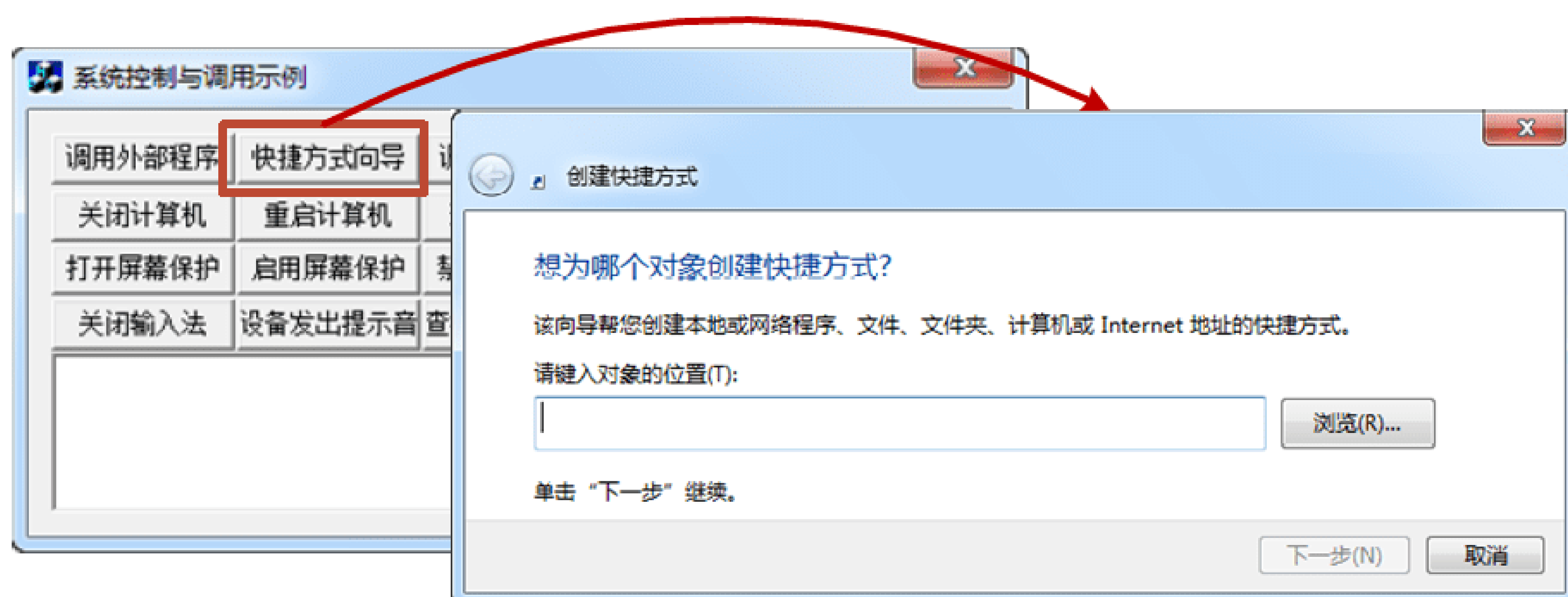


图 20-19　调用创建快捷方式向导运行效果

20.3.3　访问启动控制面板中的各项

通过 CLSID_Shell 接口类可以调用 Windows 自带的功能，ControlPanelItem()方法可以

访问系统控制面板中的各项。方法是传入控制面板中各项对应的 cpl 文件名。表 20-4 中列出了控制面板中常用的各项对应的 cpl 文件名。

表 20-4　控制面板中的各项对应的cpl文件名

功　　能	cpl 文件名	功　　能	cpl 文件名
辅助功能选项	access.cpl	声音和音频设备	mmsys.cpl
添加或删除程序	appwiz.cpl	网络连接	ncpa.cpl
日期和时间	timedate.cpl	数据源（ODBC）	odbccp32.cpl
显示	desk.cpl	打印机和传真机	main.cpl
字体	main.cpl	区域和语言选项	intl.cpl
Internet 属性	inetcpl.cpl	系统	sysdm.cpl
键盘	main.cpl	拨号规则	telephon.cpl
授权	liccpa.cpl	电源选项	ups.cpl
调制解调器	modem.cpl		

下面的代码显示了调用控制面板中辅助功能选项的代码。首先调用 CoInitialize 初始化 COM 环境，然后创建 CLSID_Shell 实例到 IShellDispatch 类型的变量中。如果创建成功，再调用 IShellDispatch 变量的 ControlPanelItem()方法，传入控制面板中各项的名称来访问控制面板中的相应项。代码如下：

```
01  //访问控制面板中的各项的实现函数
02  void CSysControlSampleDlg::OnButtonExeControlitem()
03  {
04      if (SUCCEEDED(CoInitialize(NULL)))            //初始化 COM 工作环境
05      {
06          IShellDispatch* pShellDispatch=NULL;      //定义接口指针
07          //创建 CLSID_Shell 接口实例
08          if (SUCCEEDED(CoCreateInstance(CLSID_Shell, NULL,
09       CLSCTX_INPROC_SERVER, IID_IDispatch, (void**)&pShellDispatch)))
10          {
11              //如果成功
12              COleVariant   OleVariant("sysdm.cpl");
13              //初始化控制面板对应的 cpl 名
14              //调用指定 cpl 名对应的对话框程序
15              HRESULT hResult = pShellDispatch->
16                  ControlPanelItem(OleVariant.bstrVal);
17              pShellDispatch->Release();
18          }
19          else
20              //如果创建实例失败，则显示错误信息
21              WriteLog("创建 CLSID Shell 实例失败");
22          CoUninitialize();                         //清理 COM 工作环境
23      }
24      else
25          WriteLog("初始化 Shell 失败"); //初始化 COM 环境失败，则显示错误信息
26  }
```

单击“调用控制面板”按钮，则系统会弹出控制面板中的“系统属性”对话框，如图 20-20 所示。

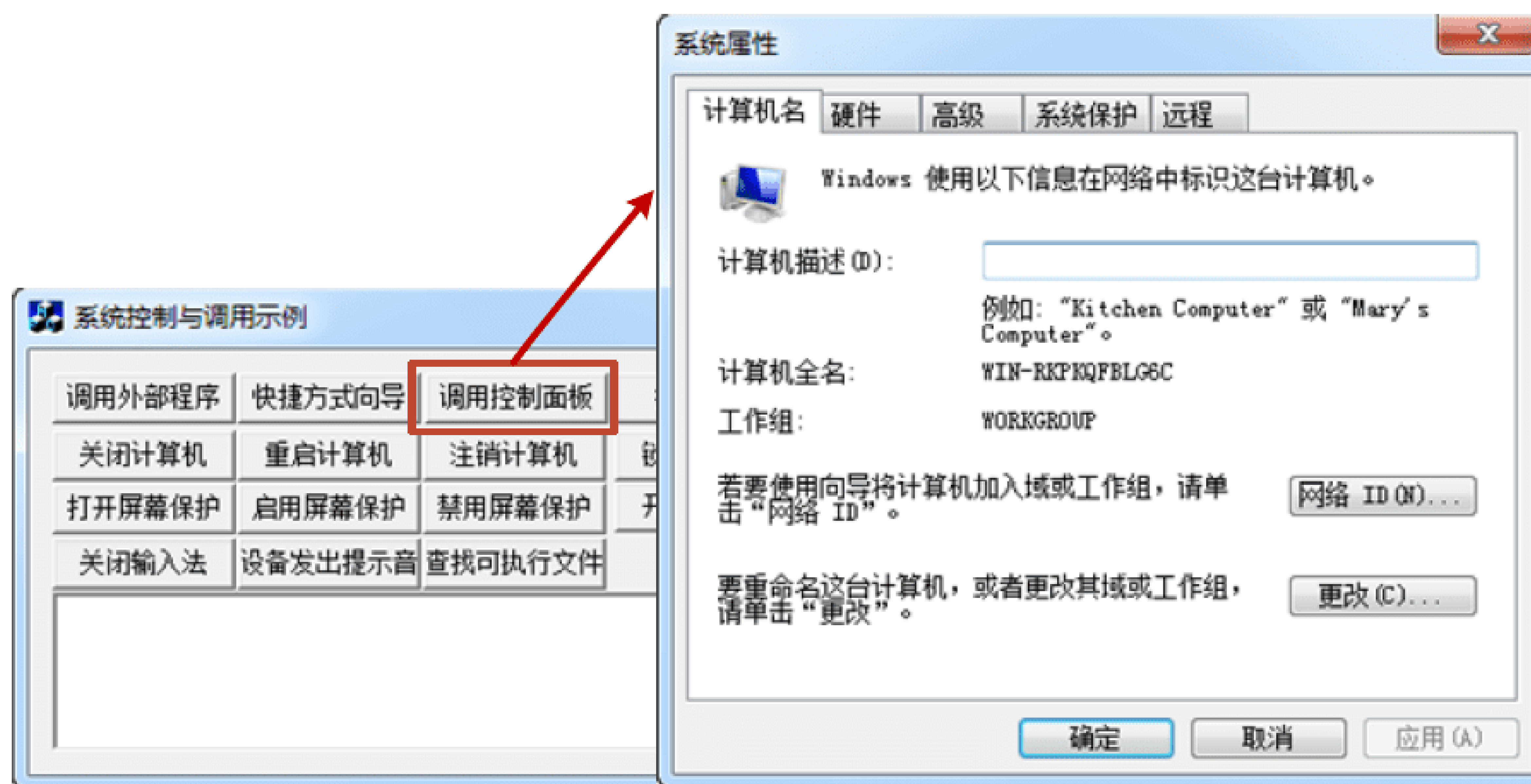


图 20-20　访问控制面板中的各项

20.3.4　控制光驱的弹开与关闭

Windows 系统提供了多媒体控制接口 MCI（Media Control Interface），实现对多媒体设备的操作。mciSendCommand()函数的功能是向多媒体设备发送命令，诸如光驱的弹开与关闭、播放音频文件等。其函数原型为：

```
MCIERROR mciSendCommand(          //向多媒体设备发送命令
  //指定接收命令消息的多媒体设备的标识符，打开消息时此参数为 NULL
  MCIDEVICEID IDDevice,
  UINT uMsg,                      //指定消息命令
  DWORD fdwCommand,               //表示命令消息的选项
  DWORD dwParam);                 //包含命令消息的参数的结构
```

如果函数成功，则返回 0；如果函数失败，则返回错误代码值，通过 mciGetErrorString()函数可以获得错误原因。下面是实现控制光驱弹开和关闭的功能代码。

```
01  void CSysControlSampleDlg::OnButtonOpenCdrom() //打开光驱实现函数
02  {
03      MCI_OPEN_PARMS   mciOpenParms;              //定义打开参数变量
04      mciOpenParms.lpstrDeviceType = "cdaudio";//初始化打开设备的类型为光驱
05      if (!mciSendCommand(NULL, MCI_OPEN, MCI_OPEN_TYPE,
06          (DWORD)(LPVOID)&mciOpenParms))//发送打开光驱命令，获取光驱对应的信息
07          //发送弹开光驱门命令
08          mciSendCommand(mciOpenParms.wDeviceID, MCI_SET,
09                         MCI_SET_DOOR_OPEN,NULL);
10  }
11  void CSysControlSampleDlg::OnButtonCloseCdrom()//弹开光驱实现函数
12  {
13      //发送弹开系统中的光驱命令
14      mciSendCommand(mciGetDeviceID("cdaudio"), MCI_SET,
15                     MCI_SET_DOOR_CLOSED,NULL);
16  }
```

上面代码首先调用 mciSendCommand()函数打开多媒体设备。然后在设备句柄的基础上执行打开或关闭的操作。如果执行弹开光驱的命令，则光驱会自动弹开；如果执行关闭

光驱的命令，则光驱会自动关闭。

20.3.5　关闭、重启、注销和锁定计算机

通过调用 Win32 API 函数 InitiateSystemShutdown()可以关闭计算机和重启计算机。其函数原型为：

```
BOOL InitiateSystemShutdown(        //退出 Windows 操作系统
  LPTSTR lpMachineName,        //表示要关闭的计算机的计算机名，为 NULL 表示关闭本机
  LPTSTR lpMessage,            //表示在关闭对话框中显示的消息
  DWORD dwTimeout,             //指定关闭消息对话框显示的时间，单位是秒
  BOOL bForceAppsClosed        //指定未被保存的应用程序是否被强制关闭
  BOOL bRebootAfterShutdown);//指定关闭计算机后是否重新启动
```

其中，关机后会显示关闭对话框，关闭对话框中会显示调用此函数的用户、由 lpMessage 参数指定的信息，并提示用户注销，同时在关闭消息对话框的显示期间，用户可以通过调用 AbortSystemShutdown()函数停止关机。此函数调用时，需要权限支持。要关闭本地计算机，调用进程必须具有 SE_SHUTDOWN_NAME 权限。要关闭远程计算机，调用进程必须在远程计算机上具有 SE_REMOTE_SHUTDOWN_NAME 的权限。默认情况下，用户在登录的计算机上具有 SE_SHUTDOWN_NAME 权限，而 Administrator 在远程计算机上具有 SE_REMOTE_SHUTDOWN_NAME 的权限。此函数失败的原因一般是计算机名称无效、无法访问或者是权限不够。

使用 ExitWindowsEx()函数也可以注销、关闭或重新启动计算机。它会发送 WM_QUERYENDSESSION 消息给所有的应用程序，以确定应用程序是否可以退出。

```
BOOL ExitWindowsEx( UINT uFlags, DWORD dwReserved);
```

其中，uFlags 参数用于指定关机类型，有效取值如表 20-5 所示。

表 20-5　关机类型标记

标　记　值	意　　义
EWX_LOGOFF	注销
EWX_POWEROFF	关闭系统并关闭电源
EWX_REBOOT	重启
EWX_SHUTDOWN	正常关机
EWX_FORCE	此标记是可选项，表示强制关机。当使用此标记，则系统在退出时，不会发送 WM_QUERYENDSESSION 和 WM_ENDSESSION 消息，使用这个可选项，应用程序会丢失数据。因此，除非在紧急情况下，不要使用此选项
EWX_FORCEIFHUNG	此标记是可选项。如果进程没有响应 WM_QUERYENDSESSION 和 WM_ENDSESSION 消息，则强制终止

在关机或注销操作时，每个应用程序应该在一定时间内终止，如果超出这个时间值，则系统会显示一个对话框，允许用户强制关闭应用程序、重试或取消关机请求。如果指定了 EWX_FORCE 值，则系统会强制关闭计算机，并且不会显示对话框。如果指定了

EWX_FORCEIFHUNG 选项，则系统会强制关闭挂起的应用程序，并且也不会显示对话框。下面显示了关闭、重启、注销和锁定计算机的代码。

```
01  void CSysControlSampleDlg::OnButtonLogoffOs()  //注销
02  {
03      ExitWindowsEx(EWX_LOGOFF,NULL);             //注销操作系统
04  }
05  void CSysControlSampleDlg::OnButtonRestartOs() //重启
06  {
07      ExitWindowsEx(EWX_REBOOT,NULL);             //重启操作系统
08  }
09  void CSysControlSampleDlg::OnButtonCloseOs()   //关机
10  {
11      ExitWindowsEx(EWX POWEROFF,NULL);           //退出操作系统
12  }
13  typedef BOOL (*LOCKFUN)(VOID);         //定义 LockWorkStation()函数原型
14  void CSysControlSampleDlg::OnButtonLockSystem() //锁定计算机
15  {
16      //装载 user32.dll
17      HINSTANCE hInstance = ::LoadLibrary("user32.dll");
18      //获取函数指针
19      LOCKFUN pFun= (LOCKFUN)::GetProcAddress(hInstance,
20                       "LockWorkStation");
21      pFun();                                     //执行函数，锁定计算机
22  }
```

在上面代码中，OnButtonLogoffOs()函数、OnButtonRestartOs()函数和 OnButtonCloseOs()函数分别实现注销计算机、重启计算机和关闭计算机。OnButtonLockSystem()函数装载 user32.dll 后，获取 LockWorkStation()函数的指针，并调用此函数。执行此函数会锁定计算机，与同时按下 Ctrl+Alt+Del 组合键的功能等同。

20.3.6　关闭和打开显示器

在编写程序时，因为能源或其他原因，有时需要控制显示器的开关。在 Windows 系统下，可以通过发送消息的方式控制。WM_SYSCOMMAND 消息主要用于处理与系统相关的命令。如果 wParam 参数指定为 SC_MONITORPOWER，则表示操作显示器电源。如果 lParam 参数为 2，则表示关闭显示电源；如果 lParam 参数为 1，则表示进入省电模式；如果 lParam 参数为–1，则表示打开显示器电源。例如，OnButtonCloseMonitor()函数发送消息关闭显示器电源，OnButtonOpenMonitor()函数发送消息打开显示器电源。代码如下：

```
01  void CSysControlSampleDlg::OnButtonCloseMonitor()//关闭显示器实现函数
02  {
03      //发送 WM_SYSCOMMAND 消息关闭显示器电源
04      ::SendMessage(GetSafeHwnd(),WM_SYSCOMMAND,SC_MONITORPOWER,2);
05  }
06  void CSysControlSampleDlg::OnButtonOpenMonitor()//打开显示器实现函数
07  {
08      //发送 WM_SYSCOMMAND 消息打开显示器电源
09      ::SendMessage(GetSafeHwnd(),WM_SYSCOMMAND,SC_MONITORPOWER,-1);
10  }
```

20.3.7　打开和关闭屏幕保护

使用 SendMessage()函数发送消息，可以打开屏幕保护和禁用屏幕保护程序。要打开屏幕保护程序，需要发送 WM_SYSCOMMAND 消息，并传递 SC_SCREENSAVE 指令。通过 SystemParametersInfo()函数传入 SPI_SETSCREENSAVEACTIVE 操作标识符可以启用或禁用屏幕保护程序。例如，OnButtonCloseScreensaver()函数、OnButtonEnableScreensaver()函数和 OnButtonOpenScreensaver()函数分别完成禁用屏幕保护程序、启用屏幕保护程序和打开屏幕保护程序的功能。代码如下：

```
01  //禁用屏幕保护程序实现函数
02  void CSysControlSampleDlg:: OnButtonDisableScreensaver ()
03  {
04      //调用 SystemParametersInfo()函数禁用屏幕保护程序
05      SystemParametersInfo(SPI_SETSCREENSAVEACTIVE, false, NULL, 0);
06  }
07  //启用屏幕保护程序实现函数
08  void CSysControlSampleDlg::OnButtonEnableScreensaver()
09  {
10      //调用 SystemParametersInfo()函数启用屏幕保护程序
11      SystemParametersInfo(SPI_SETSCREENSAVEACTIVE, true, NULL, 0);
12  }
13  //打开屏幕保护程序实现函数
14  void CSysControlSampleDlg::OnButtonOpenScreensaver()
15  {
16      //调用 SystemParametersInfo()函数打开屏幕保护程序
17      ::SendMessage(GetSafeHwnd(),WM_SYSCOMMAND,SC_SCREENSAVE,2);
18  }
```

20.3.8　关闭当前输入法

关闭当前输入设备上下文的输入法，需要使用 ImmGetContext()函数和 ImmSetOpenStatus()函数。ImmGetContext()函数获得输入法上下文，需要传入上下文所在的句柄，并返回一个 HIMC 类型的输入法句柄。通过这个句柄调用 ImmSetOpenStatus()函数，可以关闭或打开此设备上下文的输入法。代码如下：

```
01  void CSysControlSampleDlg::OnButtonCloseimm()//关闭当前输入法实现函数
02  {
03      //获取日志编辑框的输入法句柄
04      HIMC  himc = ImmGetContext(m_editLog.m_hWnd);
05      //使用 ImmSetOpenStatus()函数关闭输入法
06      ImmSetOpenStatus(himc, false);
07  }
```

上面代码调用 ImmGetContext()函数获取日志文本框中的输入法句柄后，调用 ImmSetOpenStatus()函数关闭当前的输入法。

20.3.9　让程序发出提示音

使用 MessageBeep()函数可以播放提示音，播放的提示音是在注册表中的[sounds]部分

定义的。函数原型为：

```
BOOL MessageBeep( UINT uType );
```

其中 uType 参数表示要播放的声音类型，其有效取值如表 20-6 所示。

表 20-6　系统支持的提示音

值	提　示　音	值	提　示　音
0xFFFFFFFF	计算机的标准提示音	MB_ICONHAND	系统等待
MB_ICONASTERISK	系统询问	MB_ICONQUESTION	系统问题
MB_ICONEXCLAMATION	系统信息	MB_OK	系统默认

程序播放系统询问声音代码如下：

```
01  MessageBeep(MB_ICONASTERISK);        //播放系统询问声音
```

20.3.10　列举系统中的可执行文件

使用 CFileFind 类可以查找指定目录下的符合条件的文件，并且提供访问查找到的文件的信息的接口函数。通过使用此文件和递归函数，可以列举出系统中的所有可执行文件。此处对可执行文件的判断标准是 EXE 扩展名。代码如下：

```
01  //查找系统中可执行文件的实现函数
02  void CSysControlSampleDlg::OnButtonCloseSearchfile()
03  {
04      ::SetCursor(LoadCursor(NULL, IDC_WAIT));    //设置等待光标
05      char szDrivers[MAX_PATH]={0};               //定义路径变量
06      CString szPath;                             //定义路径变量
07      //获取本地驱动器名
08      DWORD dwLen = GetLogicalDriveStrings(MAX_PATH, szDrivers);
09      for (int i = 0;i <dwLen; i++)        //循环处理驱动器名中包含的驱动器
10      {
11          if (((szDrivers[i] <='Z') && (szDrivers[i] >='A'))||
12              ((szDrivers[i]<='z') && (szDrivers[i] >='a')))
13          {
14              //如果是有效的驱动器名，则格式化
15              szPath.Format("%c:", szDrivers[i]);
16              FindExe(szPath);                 //查找指定驱动器中的可执行文件
17          }
18      }
19      ::SetCursor(LoadCursor(NULL, IDC_ARROW));//恢复箭头光标
20  }
```

上面的函数是用户单击“查找可执行文件”按钮时程序执行的代码。首先设置当前光标为等待状态，然后获取当前计算机上的所有驱动器的名称字符串，对于每个驱动器调用 FindExe()递归函数枚举其中的可执行文件。列举完成后，恢复光标的状态为箭头。下面的代码是查找可执行文件的递归函数。

```
01  //查找指定路径中的可执行文件
02  void CSysControlSampleDlg::FindExe(CString szPath)
03  {
```

```
04      CFileFind ff;                              //定义文件查找变量
05      CString szSearchPath = szPath + "\\*";     //格式化查找路径
06      BOOL bContinue = finder.FindFile(szSearchPath);
07      //启动文件查找过程，并记录结果
08      while (bContinue)                     //递归循环处理每个文件夹的子文件夹
09      {
10          if (iFileCount > 10) return;      //为了显示，只查询前 10 个文件
11          bContinue = ff.FindNextFile();    //查找下一个文件
12          if (ff.IsDirectory())             //如果当前查找到的文件为目录
13          {
14              //如果当前查找到的文件不是以"."开头
15              if (ff.GetFileName().Left(1) != ".")
16              {
17                  CString szRecuPath;       //定义存放路径名的变量
18                  //格式化文件名
19                  szRecuPath.Format("%s\\%s", szPath,ff.GetFileName());
20                  FindExe(szRecuPath);  //递归调用 FindExe()函数，查找此文件夹
21                  continue;             //继续下一个循环查找
22              }
23          }
24          //如果查找到的文件扩展名为 exe
25          if (ff.GetFileName().Right(3) == "exe")
26          {
27              iFileCount++;             //文件计数自增 1
28              //文件名写入日志文本框
29              WriteLog("%s%s", ff.GetFilePath(), ff.GetFileName());
30          }
31      }
32      ff.Close();                       //关闭文件查找对象
33  }
```

上面代码使用 CFileFind 类进行文件的查找。首先调用 FindFile()函数启动查找，路径为上一次递归传入的路径。判断是否查找到文件，如果查找到文件则进入 while 循环，并调用 FindNextFile()函数查找下一个符合条件的文件。判断文件是否是目录，如果不是目录，判断后缀名是否为 EXE，如果是，则将完整的可执行文件路径显示在日志文本框中；如果查找到的文件是文件夹，则判断是否是以点号开头。如果是，则表示它是真实的文件夹，则递归调用函数本身 FindExe()。执行完后，继续进行循环，直到所有的文件都遍历完成。因为文本框的字数是有限制的，所以此处只查询前 10 个可执行文件。程序运行效果如图 20-21 所示。

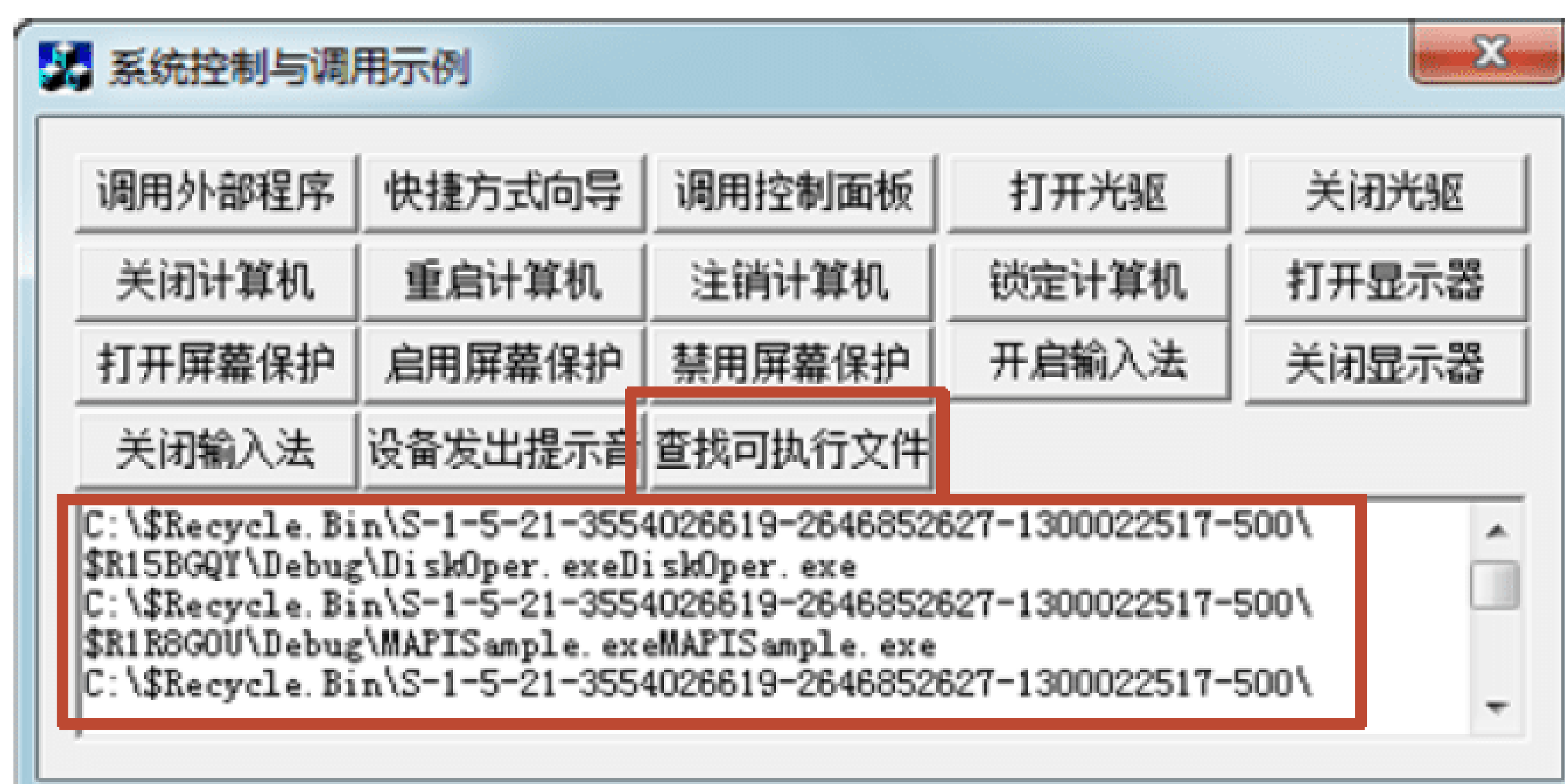


图 20-21　列举系统中的可执行文件运行效果

20.4　应用程序操作

本节介绍有关应用程序操作的技巧，包括如何判断程序的状态、禁止程序重复运行、检索任务列表、判断程序是否运行、如何在程序中运行其他应用程序、如何修改其他进程中的对话框标题和图标、如何设计换肤程序、PE 档案格式分析以及列举程序使用的 DLL 文件等。

20.4.1　禁止程序重复运行

要禁止程序重复运行有多种方法，此处使用互斥对象实现，在程序运行启动的 InitInstance()初始化实例函数中创建命名互斥对象，如果返回值为 ERROR_ALREADY_EXISTS 表示系统已经存在此名称的互斥对象，说明已经有程序实例在运行，程序会返回。在 InitInstance()函数退出前，运行 CloseHandle()函数关闭互斥对象，则关闭程序后，再次运行程序时，此名称的互斥对象已经不存在，可以继续运行，程序代码如下：

```
01  const char* MyClassName = "CAppOperSampleDlg"; //定义类名常量
02  BOOL CAppOperSampleApp::InitInstance()         //类的初始化实例函数
03  {
04      //创建指定名称的关键段句柄
05      HANDLE hMutex = CreateMutex(NULL, true, MyClassName);
06      //如果重复运行，则返回错误值 ERROR_ALREADY_EXISTS
07      if(GetLastError() == ERROR_ALREADY_EXISTS)
08         return false;
09  }
```

在 CAppOperSampleApp 应用程序类的 InitInstance()函数中，使用 CreateMutex()函数创建名称为 MyClassName 的变量存储的值为 CAppOperSampleDlg 的互斥对象，并判断返回值是否为 ERROR_ALREADY_EXISTS。如果是，则退出应用程序实例，否则继续执行。在 InitInstance()函数返回前，调用 CloseHandle()关闭应用程序的实例互斥对象句柄。这样，当运行程序后，再次运行程序时，不会启动新的程序实例。程序运行效果如图 20-22 所示。

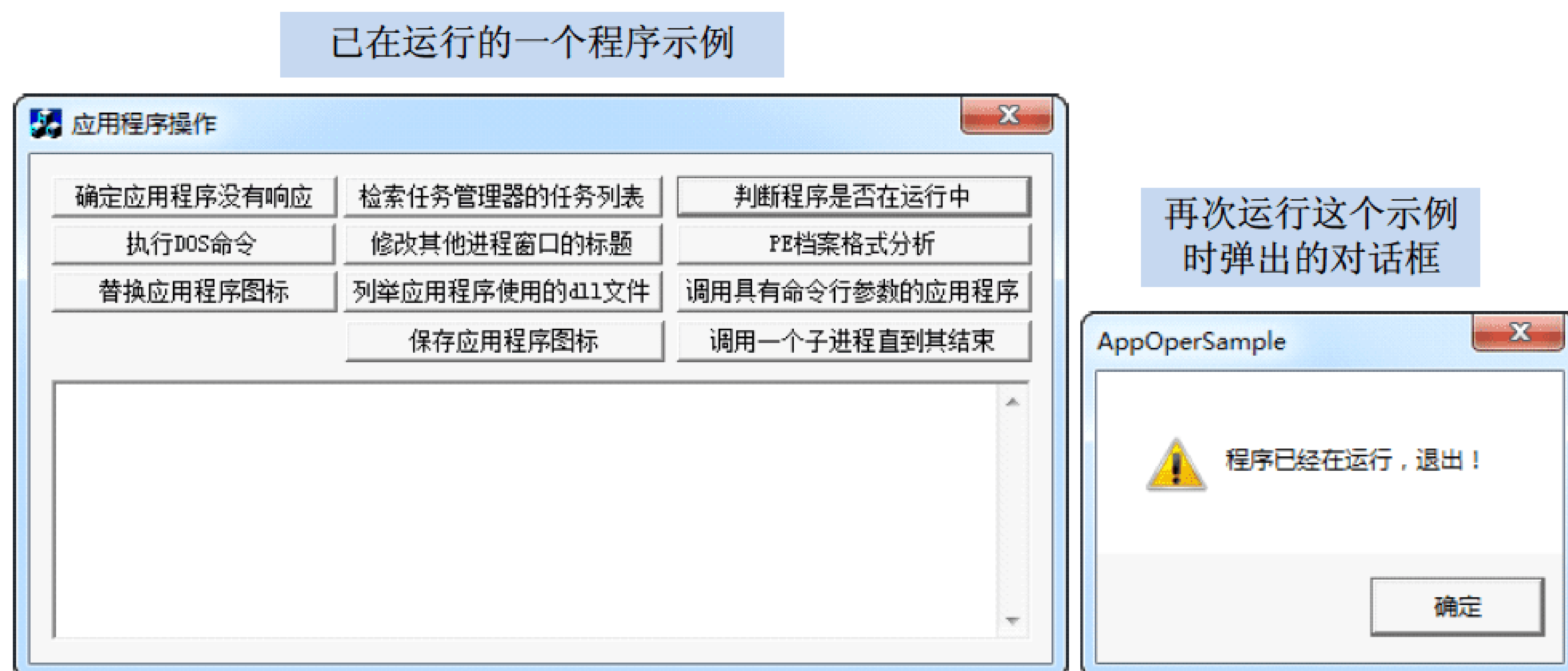


图 20-22　禁止程序重复运行效果

20.4.2　如何确定应用程序没有响应

在 User32.dll 动态库中，提供了 IsHungAppWindow()函数，用于检测程序是否没有响应。因为此 API 不是公开的，因此，需要使用 GetProcAddress()函数获取函数指针，使用函数指针调用此函数，此函数的参数就是要判断的程序的句柄。以下代码判断记事本程序是否没有响应。

```
01  typedef BOOL  (WINAPI *PROCISHUNGAPPWINDOW)(HWND); //定义函数原型
02  //确定应用程序是否没有响应的函数
03  void CAppOperSampleDlg::OnButtonIsResponse()
04  {
05      //定义 IsHungAppWindow()函数指针
06      PROCISHUNGAPPWINDOW  IsHungAppWindow;
07      PROCISHUNGTHREAD  IsHungThread;     //定义 IsHungThread()函数指针
08      HMODULE hUser32=GetModuleHandle("user32"); //获取 user32 模块句柄
09      IsHungAppWindow = (PROCISHUNGAPPWINDOW)
10          GetProcAddress(hUser32,"IsHungAppWindow");
11      //获取 IsHungAppWindow()函数指针
12      CString szAppName = "Notepad";              //定义记事本进程名称
13      HWND hWnd = ::FindWindow(szAppName, NULL);//查找记事本程序的窗口句柄
14      if (hWnd)                                   //如果窗口句柄有效
15      {
16          //调用 IsHungAppWindow()函数判断应用程序窗口是否没有响应
17          if (IsHungAppWindow(hWnd))
18              //程序没有响应
19              WriteLog("%s 程序没有响应", szAppName);
20          else
21              WriteLog("%s 程序有响应", szAppName);   //程序有响应
22      }
23      else
24          WriteLog("%s 程序没有运行", szAppName);    //程序没有启动
25  }
```

上面代码首先调用 GetModuleHandle()函数装载 user32.dll 运行库，然后调用 GetProcAddress()函数获取 IsHungAppWindow()函数的函数指针。使用 FindWindow()函数查找记事本程序的句柄，通过 IsHungAppWindow()函数指针传入记事本程序的句柄，判断程序是否有响应。最后根据情况，将程序是否有响应的信息显示在日志编辑框中。程序运行效果如图 20-23 所示。

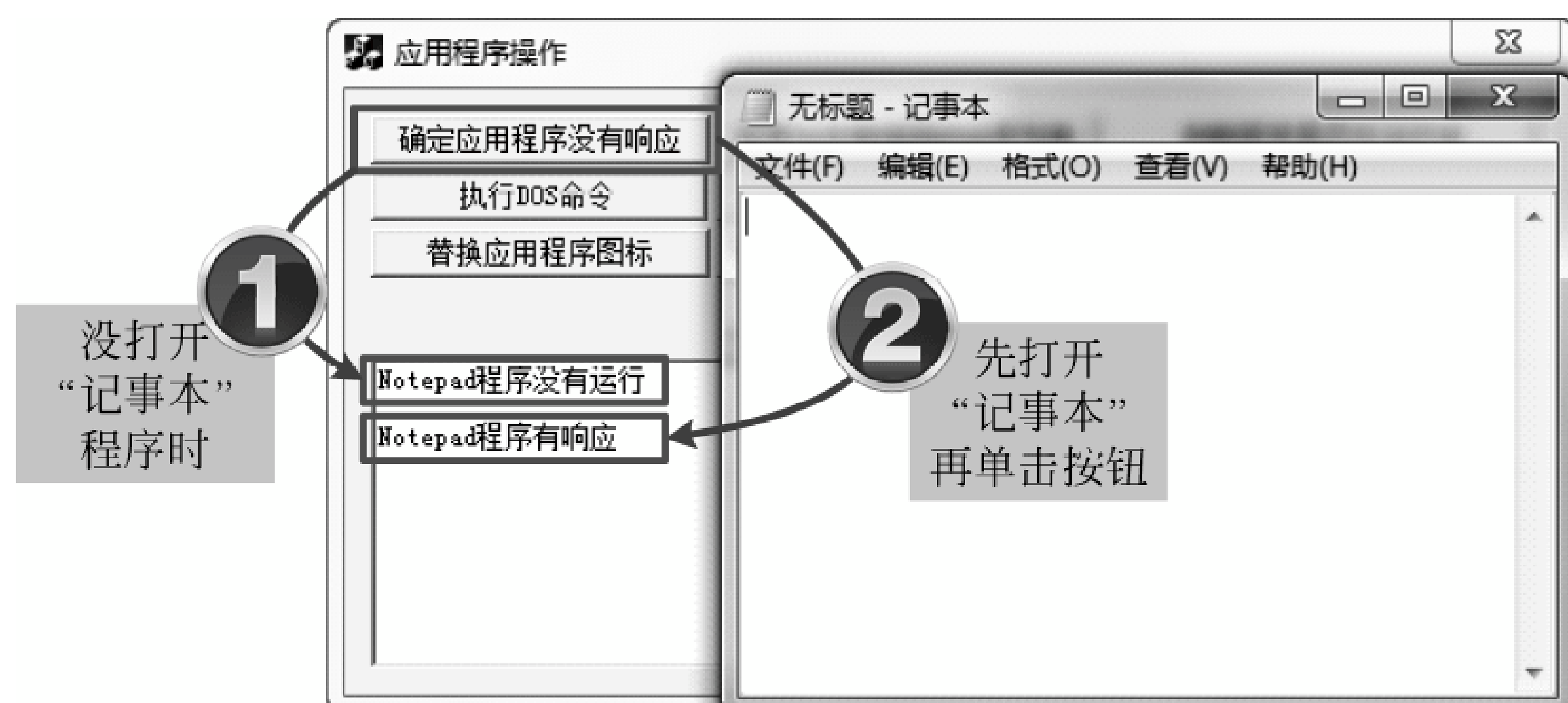


图 20-23　确定程序是否有响应运行效果

20.4.3　检索任务管理器中的任务列表

调用 EnumWindows()函数可以枚举当前系统中屏幕上所有的顶层级别对话框，并且通过回调函数，处理每次枚举出来的对话框信息。其函数原型为：

```
BOOL EnumWindows(
  WNDENUMPROC lpEnumFunc,    //指向回调函数的函数指针
  LPARAM lParam);            //应用程序自定义的传递给回调函数的值
```

EnumWindowsProc()回调函数是应用程序为 EnumWindows()函数指定的在每次枚举成功一次后所执行的函数。其函数原型为：

```
BOOL CALLBACK EnumWindowsProc(
  HWND hwnd,         //函数接收到的顶层对话框句柄
  LPARAM lParam );   //从 EnumWindows()函数传递过来的应用程序自定义数据
```

WNDENUMPROC 类型定义了回调函数的指针类型。EnumWindowsProc 是应用程序自定义函数名的替代名称。要继续进行枚举，则回调函数必须返回 true，要停止枚举，回调函数必须返回 false。具体代码如下：

```
01  void CAppOperSampleDlg::OnButtonListtask() //列出任务列表中的任务
02  {
03      //调用枚举窗口的函数
04      ::EnumWindows((WNDENUMPROC)enumProcFunc,(LPARAM)this);
05  }
```

上面代码调用 EnumWindows()函数启动枚举对话框的过程，指定回调函数为 enumProcFunc()，并将当前对话框的句柄传递给回调函数。下面是回调函数的处理代码。

```
01  BOOL CALLBACK CAppOperSampleDlg::enumProcFunc(HWND hWnd,
02                                                 LPARAM lParam)
03  {
04      //枚举处理函数
05      CAppOperSampleDlg * pDlg = (CAppOperSampleDlg*)lParam;
06      //根据参数获取对话框对象
07      TCHAR szTitle[MAX_PATH] = {0};          //定义标题变量
08      if (hWnd == NULL)
09          return false;
10      if (hWnd == pDlg->m_hWnd)
11          return true;
12      //如果句柄表示的是可视的对话框，则
13      if (::IsWindow(hWnd) && ::IsWindowVisible(hWnd)
14           && ((GetWindowLong(hWnd, GWL_EXSTYLE)&WS_EX_TOOLWINDOW)
15          !=WS_EX_TOOLWINDOW)
16           &&(GetWindowLong(hWnd, GWL_HWNDPARENT)==0))
17      {
18          memset(szTitle, 0x00, sizeof(szTitle)); //初始化标题变量
19          ::GetWindowText(hWnd, szTitle, sizeof(szTitle));
20          //获取窗口的标题
21          if (strlen(szTitle) == 0)
22              return true;
23          DWORD dwProcessID = 0;                    //定义进程 ID 变量
24          //获取线程对应的进程 ID
25          ::GetWindowThreadProcessId(hWnd,&dwProcessID);
```

```
26            //将任务信息显示在文本框中
27            WriteLog("%s \t\t 进程 ID=%d\r\n", szTitle, dwProcessID);
28        }
29        return true;                                    //函数返回，继续枚举
30    }
```

上面代码将回调函数的 hWnd 参数转换成对话框类 CAppOperSampleDlg。判断枚举到的对话框是否是正常显示状态，如果是，则通过 GetWindowText()函数获取对话框的标题。通过 GetWindowThreadProcessId()函数获取对话框对应的进程 ID，并将对话框的标题和进程 ID 显示在程序的文本框中。程序运行效果如图 20-24 所示。

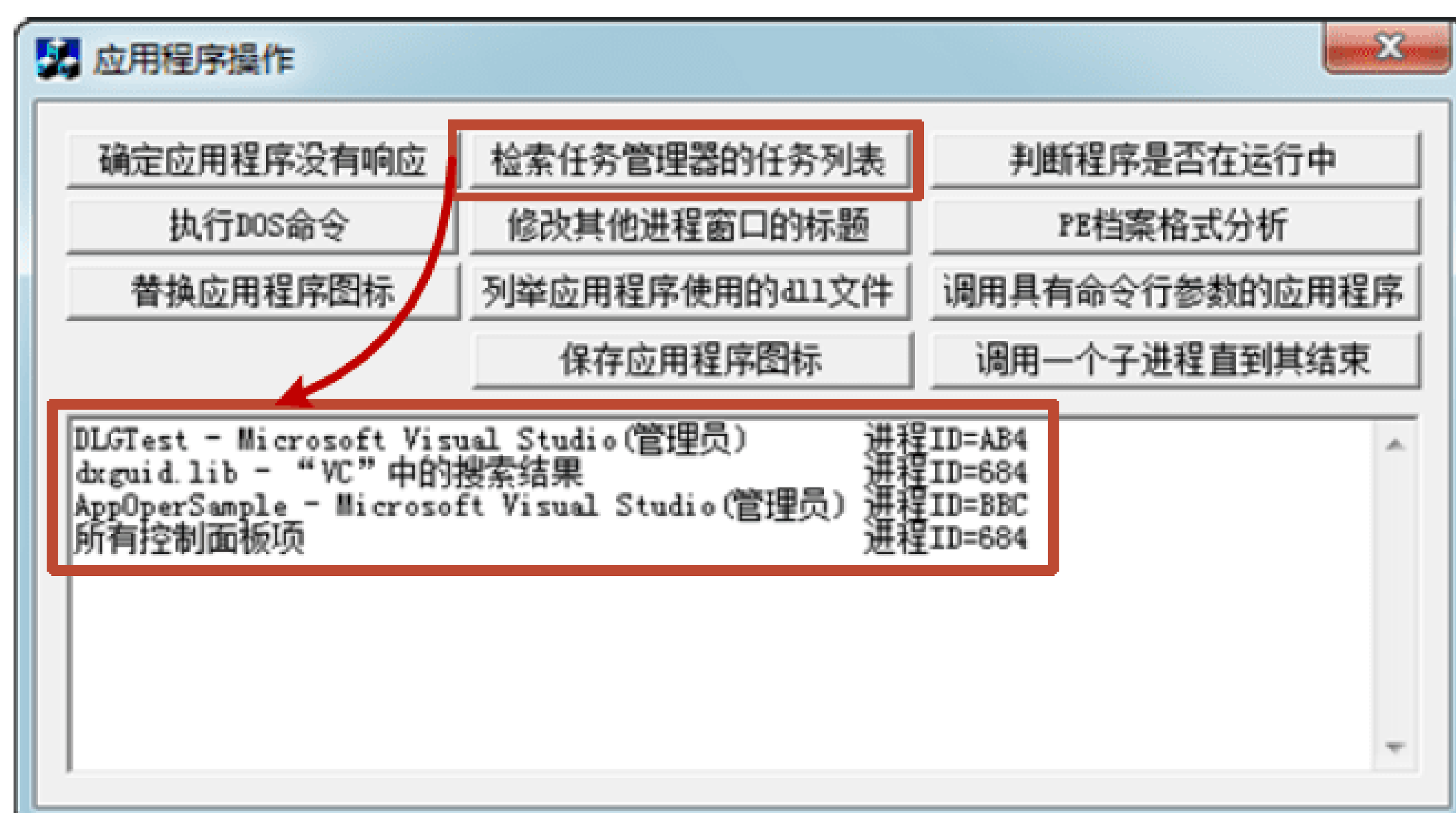


图 20-24　检索任务管理器中的任务列表运行效果

20.4.4　判断某个程序是否运行

要判断某个程序是否在运行，可以通过 Windows 中提供的快照系列，在进程快照中依次检索每个进程。Windows API 函数 CreateToolhelp32Snapshot()可以创建进程和线程等信息的快照。调用 Process32First()函数，可以检索指定进程快照中的第一进程的信息。调用 Process32Next()函数检索指定进程中的下一个进程信息。Process32Next()函数需要与 Process32First()函数搭配使用。下面的代码用于判断记事本程序是否运行。

```
01    //判断记事本程序是否在运行的实现函数
02    void CAppOperSampleDlg::OnButtonIfRunning()
03    {
04        BOOL bRunning = false;                    //定义变量标识程序是否正在运行中
05        CString szAppName="C:\\Windows\\Notepad.exe";  //定义应用程序名称
06        HANDLE   hPS;                             //进程快照句柄
07        PROCESSENTRY32   pe;                      //进程条目变量
08        //创建进程快照
09        hPS = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
10        if (hPS==INVALID_HANDLE_VALUE)
11            return;                               //创建失败，返回
12        memset(&pe,0,sizeof(pe));                 //初始化 PROCESSENTRY32 结构
13        pe.dwSize=sizeof(PROCESSENTRY32);         //为 dwSize 赋值
14        if (Process32First(hPS, &pe))             //检索第一个进程
15        {
16            do
17            {
```

```
18              //比较是否是要判断的应用程序
19              if (szAppName.CompareNoCase(pe.szExeFile))
20              {
21                  bRunning = true;            //如果是，则赋值变量
22                  break;                      //退出循环
23              }
24          }
25          while(Process32Next(hPS,&pe));      //检索下一个进程
26      }
27      CloseHandle(hPS);                       //关闭快照句柄
28      if (bRunning)
29          //输出成功信息
30          WriteLog("程序%s 正在运行......", szAppName);
31      else
32          WriteLog("程序%s 没有运行。", szAppName);//输出失败信息
33  }
```

上面的代码首先调用 CreateToolhelp32Snapshot()函数创建 Windows 进程快照，然后调用 Process32First()函数和 Process32Next()函数依次检索进程快照中的每个进程的信息。如果进程的可执行文件路径名与判断的程序名称相同，则表示程序正在运行中。程序运行效果如图 20-25 所示。

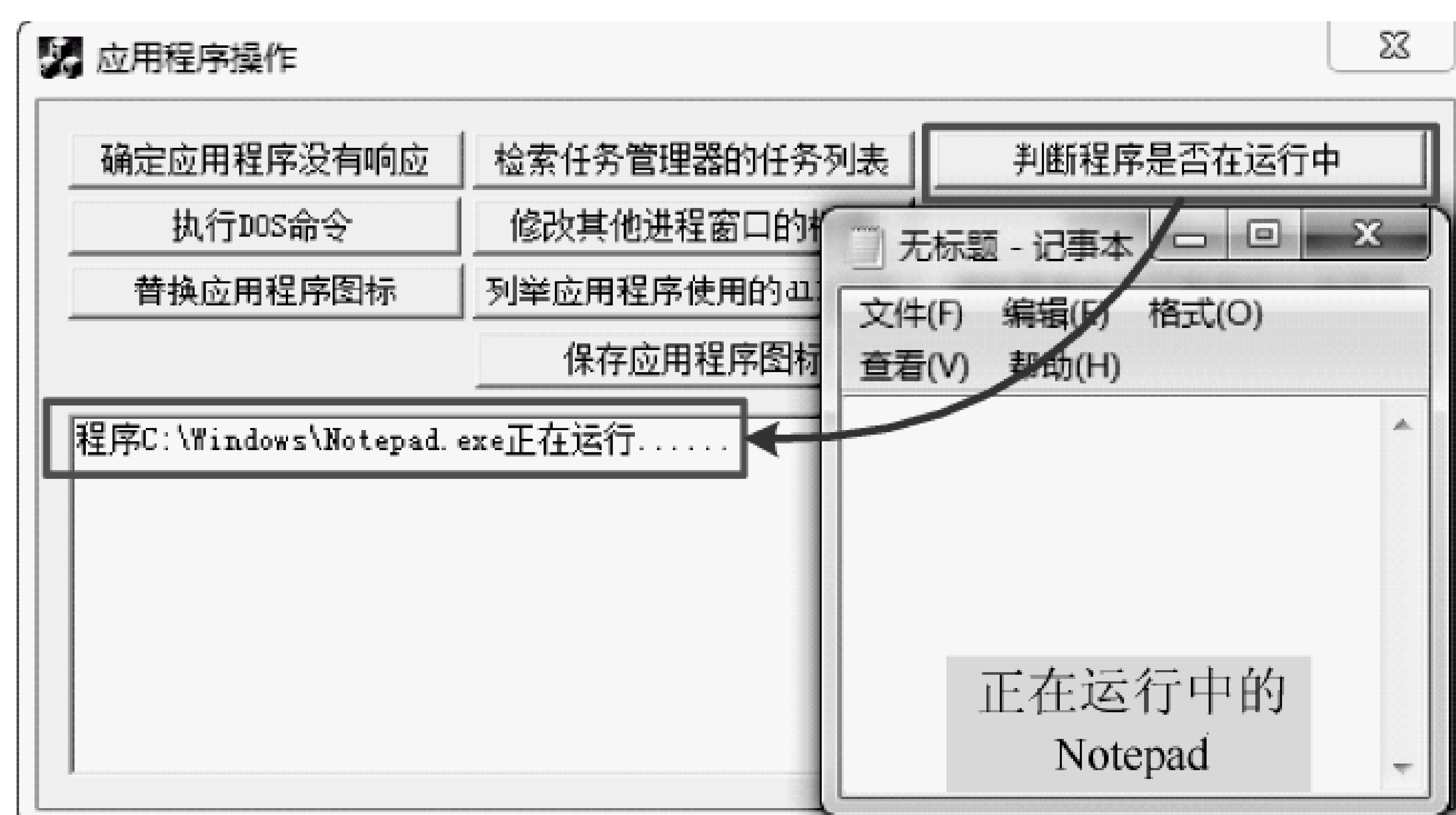


图 20-25　判断某个程序是否运行的效果

20.4.5　怎样在程序中执行 DOS 命令

调用 system()函数可以执行 DOS 命令。其函数原型为：

```
int system( const char *command );          //要执行的 DOS 命令
int _wsystem( const wchar_t *command );     //要执行的 DOS 命令
```

如果 command 参数为 NULL，并且查找到命令解释器，则函数返回非 0 值。如果没有查找到命令解析器，则返回 0，并且错误代码为 ENOENT；如果 command 参数不为 NULL，则函数返回命令解析器返回的值。如果返回值为–1，表示执行时发生错误。以下代码执行 copy 命令，将 C 盘根目录下的 1.txt 文件复制到 D 盘根目录下。

```
01  void CAppOperSampleDlg::OnButtonRunCopydos()
02  {
03      system("copy C:\\1.txt D:\\");
04  }
```

20.4.6　修改其他进程中对话框的标题

使用 FindWindow()函数和 SetWindowText()函数可以修改其他进程中对话框的标题，FindWindow()函数在 20.4.2 小节中介绍过，SetWindowText()函数可以修改指定对话框的标题，其原型为：

```
BOOL SetWindowText(
  HWND hWnd,                                    //要修改标题的对话框的句柄
  LPCTSTR lpString );                           //要设置的对话框的标题
```

下面的代码用于修改打开的记事本对话框的标题。

```
01  void CAppOperSampleDlg::OnButtonUpdateTitle()
02  {
03      HWND hWnd = ::FindWindow("Notepad", NULL);
04      if (hWnd)
05          ::SetWindowText(hWnd,
06                "这是从 AppOperSampleDlg 程序中修改后的标题");
07  }
```

在上面代码中，使用 FindWindow()函数获取记事本对话框的句柄，然后调用 SetWindowText()函数设置打开的记事本对话框的标题为“这是从 AppOper- SampleDlg 程序中修改后的标题”。程序运行效果如图 20-26 所示。



图 20-26　修改其他进程中对话框的标题运行效果

20.4.7　如何设计换肤程序

要使应用程序支持换肤功能，则一般通过动态装载位图，并将位图作为工具条的背景即可。下面的代码是支持换肤程序的 OnCreate()函数需要修改的地方。其中 m_wndReBar 对象的 AddBar()函数将添加工具栏，并且背景色可以组合在一起。在 OnCreate()函数的结尾处，调用应用程序自定义函数 LoadFace()实现换肤。代码如下：

```
01  int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
02  {
03      if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
04          return -1;
```

```
05      ……      //此处代码省略
06      if (!m_wndReBar.Create(this) ||
07          !m_wndReBar.AddBar(&m_wndToolBar,NULL,NULL,
08          RBBS_GRIPPERALWAYS|RBBS_FIXEDBMP|RBBS_BREAK))
09      {
10          TRACE0("Failed to create rebar\n");
11          return -1;      //fail to create
12      }
13      ……      //此处代码省略
14      LoadFace("back.bmp");
15      return 0;
16  }
```

下面的代码是实现换肤功能的用户自定义函数。其中，bkPath 参数是要作为皮肤背景色的位图的文件名。函数首先调用 LoadImage()函数装载位图，然后调用 m_wndReBar 变量的 GetReBarCtrl()函数获取工具栏控件对象，并设置 REBARBANDINFO 结构的 fMask 成员变量为 RBBIM_BACKGROUND，用于表示位图会作为背景色。最后调用 SetBandInfo()函数设置工具栏信息参数，并调用 UpdateWindow()函数更新对话框的显示。代码如下：

```
01  void CMainFrame::LoadFace(CString bkPath)
02  {
03      HBITMAP m_bmpBack=(HBITMAP)LoadImage(AfxGetInstanceHandle(),
04        bkPath, IMAGE_BITMAP,0,0,LR_LOADFROMFILE|LR_CREATEDIBSECTION);
05      CReBarCtrl& rc=m_wndReBar.GetReBarCtrl();
06      REBARBANDINFO ri;
07      memset(&ri,0,sizeof(REBARBANDINFO));
08      ri.cbSize=sizeof(ri);
09      ri.fMask=RBBIM_BACKGROUND;
10      if (m_bmpBack != INVALID_HANDLE_VALUE)
11          ri.hbmBack=m_bmpBack;
12      else
13          ri.hbmBack = NULL;
14      rc.SetBandInfo(0,&ri);
15      rc.UpdateWindow();
16  }
```

程序初始情况下，会装载 back.bmp 文件作为工具栏的背景色。程序运行效果如图 20-27 所示。



图 20-27　换肤程序运行效果

注意：本程序的功能实现是依赖 Visual C++ 6.0 安装时在系统文件夹下加入的库 MSVCRTD.DLL 及其他库，但 Visual Studio 2010 安装时没有包括这些库，所以

本工程在 Visual Studio 2010 中编译后没有修改工具栏肤色的效果，需要在 Visual C++ 6.0 下编译。

20.4.8　PE 档案格式分析

PE 档案格式是 Windows 应用程序使用的档案格式，通过分析 PE 档案格式，可以获取应用程序的多种信息。使用 MapAndLoad()函数可以映射档案文件，并从档案文件中预载数据。其函数原型为：

```
BOOL MapAndLoad(
    IN LPSTR ImageName,      //指定要载入的档案文件的名称
    IN LPSTR DllPath,        //指定定位档案文件的路径
    OUT PLOADED_IMAGE LoadedImage,
                             //PLOADED_IMAGE 结构指针，存储档案文件信息
    IN BOOL DotDll,          //指定默认的扩展名是否为 DLL
    IN BOOL ReadOnly );      //指定访问模式是否是只读
```

调用此函数映射装载 PE 文件后，使用完时，需要调用 UnMapAndLoad()函数释放映射，参数是 MapAndLoad()函数返回的 PLOADED_IMAGE 类型的变量。以下代码分析了 user32.dll 的 PE 档案格式。

```
01  void CAppOperSampleDlg::OnButtonPeparser()     //PE 档案格式分析
02  {
03      char szFile[MAX_PATH]={0};                 //要解析的 PE 文件名
04      strcpy(szFile, "user32.dll");              //解析 user32.dll 文件
05      LOADED_IMAGE li;                           //信息结构变量
06      if (!MapAndLoad(szFile, 0, &li, false, true))//装载 PE 文件
07      {
08          WriteLog("PE 档案格式分析--MapAndLoad 错误");//装载失败，显示错误
09                                                     //信息
10          return;                                //并返回
11      }
12      //分析装载的 PE 映射的基本信息
13      WriteLog("模块名称=%s\r\n 文件句柄=%d\r\n 映射地址=%08X\r\n 字符集=
14           %08X\r\n 档案大小=%d\r\n 映射%s 是内核模式下的可执行映射\r\n 映射
15           %s 是 16 位可执行映射\r\nLinks=%08X-%08X\r\n
16          映射大小=%u\r\n",li.ModuleName, li.hFile,li. MappedAddress,
17           li.Characteristics, li.SizeOfImage,li.fSystemImage ?
18          "" : "不", li.fDOSImage ? "" : "不", li.Links.Blink,
19          li.Links.Flink, li.SizeOfImage);
20      PIMAGE_NT_HEADERS ntHeader = li.FileHeader;//获取 NT 头结构变量
21      //输出 NT 头信息
22      WriteLog("NT 头信息 FileHeader=%08X\r\n", ntHeader->FileHeader);
23      //输出 COFF 段数量
24      WriteLog("共有%d 个 COFF 个段头", li.NumberOfSections);
25      //循环输出每段信息
26      for (int i = 0;i < (int)li.NumberOfSections;i++)
27      {
28          //获取第 i 个段
29          PIMAGE_SECTION_HEADER psHeader = (PIMAGE_SECTION_HEADER)&li.
30              Sections[i];
31          //输出段名称
```

```
32          WriteLog("第%d 个 COFF 个段头信息    名称=%s", i, psHeader->Name);
33      }
34      if (UnMapAndLoad(&li))
35          //卸载档案文件
36          WriteLog("成功卸载%s 档案文件", szFile);
37      else
38          WriteLog("卸载%s 档案文件失败", szFile);
39  }
```

上面的代码分析了 user32.dll 动态库的 PE 档案格式。首先调用 MapAndLoad()函数映射并装载指定的动态库，通过返回的 PLOADED_IMAGE 类型的变量获取相应的信息。最后调用 UnMapAndLoad()函数卸载 DLL，并将获取的信息显示在日志文本框中。程序的运行效果如图 20-28 所示。

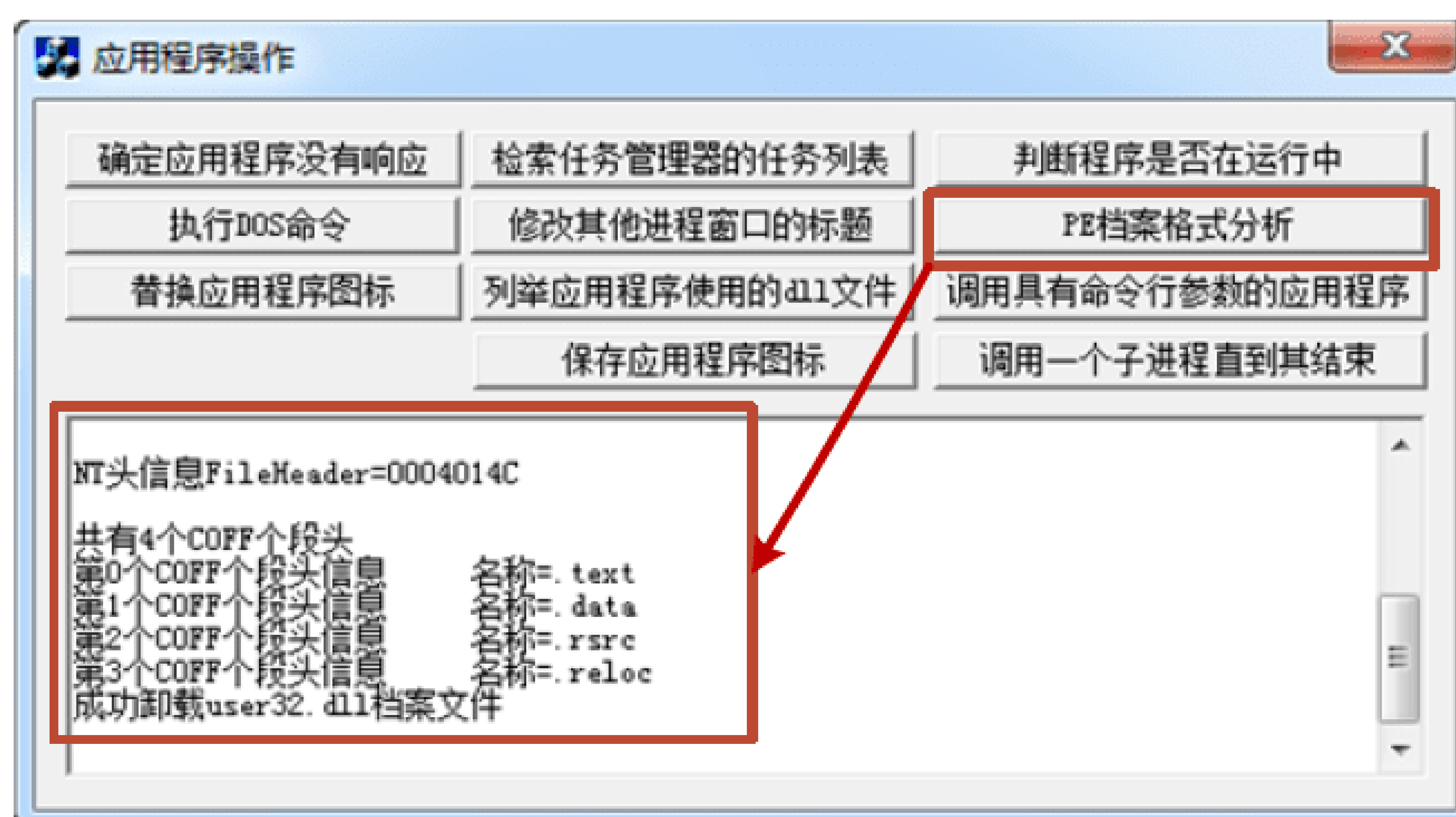


图 20-28　PE 档案格式分析运行效果

从上面的运行结果可以看出，user32.dll 共有 4 个 COFF 段，分别为.text、.data、.rsrc 和.reloc 段，其中还包括其他相关信息。

20.4.9　修改应用程序图标

要修改应用程图标，首先需要使用 LoadIcon()函数装载替换后的图标，然后需要调用 FindWindow()函数查找应用程序的句柄，最后向要替换图标的应用程序发送 WM_SETICON 消息，替换图标，WM_SETICON 消息的第一个参数表示图标的规格，第二个参数表示要替换的图标的句柄。具体代码如下：

```
01  void CAppOperSampleDlg::OnButtonReplaceicon()  //修改应用程序图标
02  {
03      //装载图标资源
04      HICON hIconNew=AfxGetApp()->LoadIcon(IDI_ICON_TEST);
05      if(hIconNew != NULL)                         //如果装载图标资源成功
06      {
07          HWND hWnd = ::FindWindow("Notepad", NULL);//查找记事本程序
08          if (hWnd != NULL)                        //如果查找到，则修改图标
09              ::SendMessage(hWnd, WM_SETICON, ICON_SMALL,
10                              (LPARAM)hIconNew);
11      }
12  }
```

在上面代码中，替换记事本程序的图标为当前程序中的 IDI_ICON_TEST 资源的图标。

运行效果如图 20-29 所示。

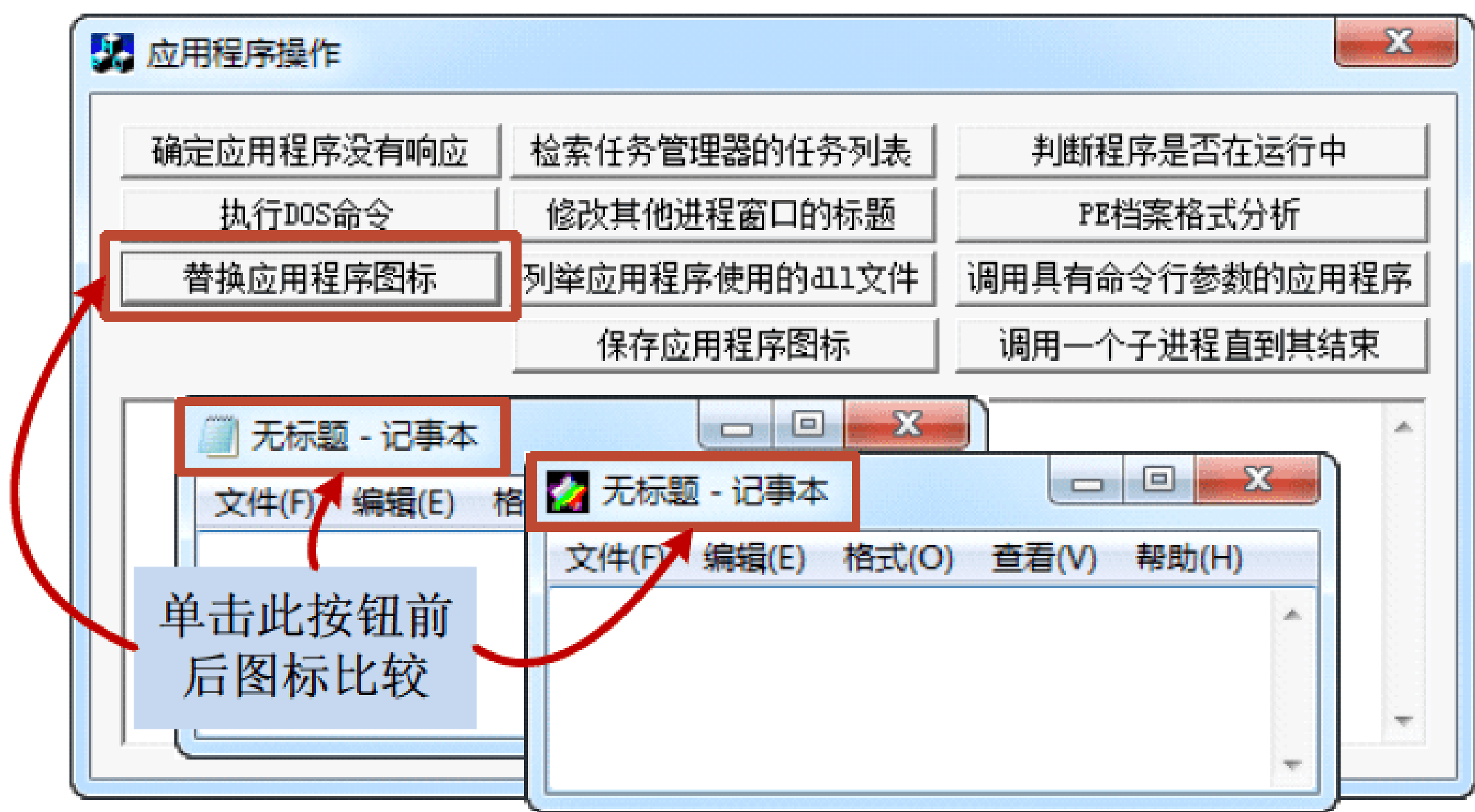


图 20-29　修改应用程序图标运行效果

20.4.10　列举应用程序使用的 dll 文件

使用 Windows 提供的快照系列函数可以获取当前运行的 Win32 应用程序的信息，可以使用 Win32 宿主工具创建代码流，从而实现调试。本小节使用其中的 CreateToolhelp32Snapshot()函数、Module32First()函数和 Module32Next()函数列举出应用程序使用的 dll 文件。其中 CreateToolhelp32Snapshot()函数可以快照进程的堆栈、模块和进程使用的线程等信息，其原型为：

```
HANDLE WINAPI CreateToolhelp32Snapshot(
  DWORD dwFlags,                   //快照中包含的系统信息的部分
  DWORD th32ProcessID );           //指定进程 ID 号，如果为 NULL，则指当前进程
```

如果打开快照成功，则返回快照句柄，否则返回–1。其中，dwFlags 参数用于指定快照中包含的系统信息的部分，有效取值如表 20-7 所示。

表 20-7　快照中包含的系统信息的可用部分

值	含　　义
TH32CS_INHERIT	指示快照句柄是可以继承的
TH32CS_SNAPALL	是 TH32CS_SNAPHEAPLIST 选项、TH32CS_SNAPMODULE 选项、TH32CS_SNAPPROCESS 选项和 TH32CS_SNAPTHREAD 选项的组合
TH32CS_SNAPHEAPLIST	在快照中包含指定进程的堆栈列表
TH32CS_SNAPMODULE	在快照中包含指定进程的模块列表
TH32CS_SNAPPROCESS	在快照中包含指定进程的进程列表
TH32CS_SNAPTHREAD	在快照中包含指定进程的线程列表

Module32First()函数和 Module32Next()函数可以获取进程使用的第一个模块和下一个有效模块。其函数原型为：

```
BOOL WINAPI Module32First(
  HANDLE hSnapshot,             //CreateToolhelp32Snapshot()返回的快照句柄
  LPMODULEENTRY32 lpme);        //指向 MODULEENTRY32 结构的模块信息结构变量
BOOL WINAPI Module32Next(
  HANDLE hSnapshot,             //CreateToolhelp32Snapshot()返回的快照句柄
```

```
  LPMODULEENTRY32 lpme );    //指向 MODULEENTRY32 结构的模块信息结构变量
```

下面的代码演示了如何使用上面 3 个函数列举出应用程序调用的 DLL。

```
01  //列举应用程序使用的 dll 文件
02  void CAppOperSampleDlg::OnButtonExeusedlls()
03  {
04      //创建进程使用的模块快照
05      HANDLE hModule = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE,0);
06      if (hModule == NULL)                              //如果创建失败
07      {
08          //显示错误信息
09          WriteLog("调用 CreateToolhelp32Snapshot()函数失败");
10          return;                                       //并返回
11      }
12      WriteLog("当前进程调用的 dll 文件有:");              //显示提示信息
13      MODULEENTRY32 me;                                 //定义模块信息变量
14      //获取快照中的第一个模块
15      BOOL bResult = Module32First(hModule,  &me);
16      while (bResult)                                   //循环处理模块
17      {
18          WriteLog(me.szExePath);                       //输出模块文件名
19          bResult = Module32Next(hModule,  &me);        //获取下一个模块
20      }
21      CloseHandle(hModule);                             //关闭模块快照句柄
22  }
```

上面代码首先调用 CreateToolhelp32Snapshot()函数获取当前进程的模块快照句柄。然后调用 Module32First()函数查询当前进程使用的第一个模块。如果返回值为 true，表示查询到的模块值是有效的，如果返回 false，则退出函数。接下来执行 while 循环，使用 Module32Next()函数继续查询进程使用的下一个模块。最后将查询到的所有模块信息显示在日志文本框中。程序运行效果如图 20-30 所示。

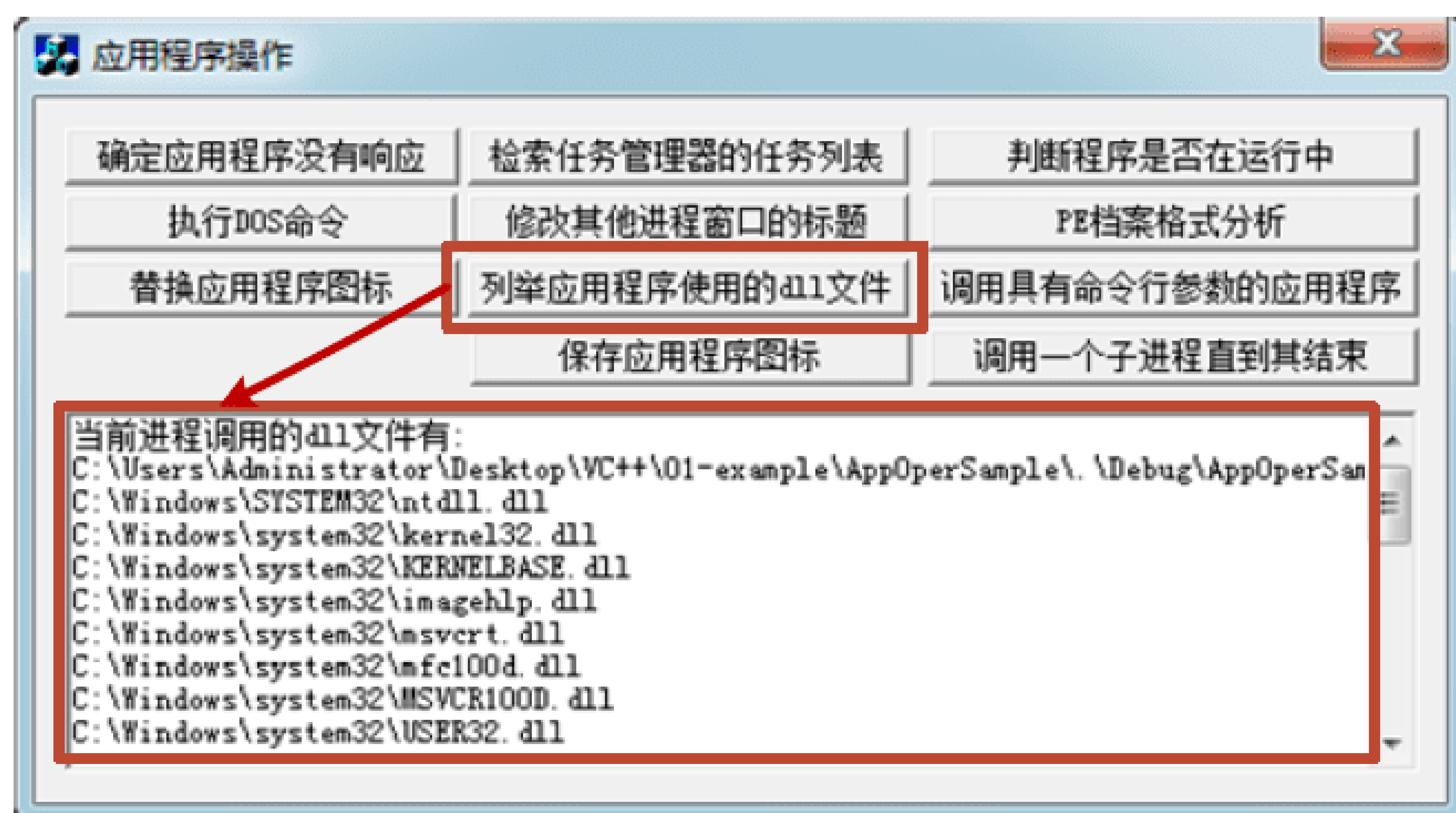


图 20-30　列举应用程序使用的 dll 文件运行效果

20.4.11　调用具有命令行参数的应用程序

在应用程序中要调用具有命令行参数的应用程序，可以使用 WinExec()函数，其函数原型为：

```
UINT WinExec(
  LPCSTR lpCmdLine,          //执行的命令行命令，参数包含在命令行字符串之中
```

```
UINT uCmdShow);              //执行命令行程序时是否显示对话框
```

下面是执行自定义的带参数的命令行程序 AddSample 的代码。

```
01  void CAppOperSampleDlg::OnButtonExecCommand()
02  {
03      WinExec("AddSample.exe 4 6", SW_SHOWNORMAL);
04  }
```

上面的代码执行 AddSample 程序，输入的参数值是 4 和 6，函数执行的功能是将两个数相加并输出，程序运行的效果如图 20-31 所示。

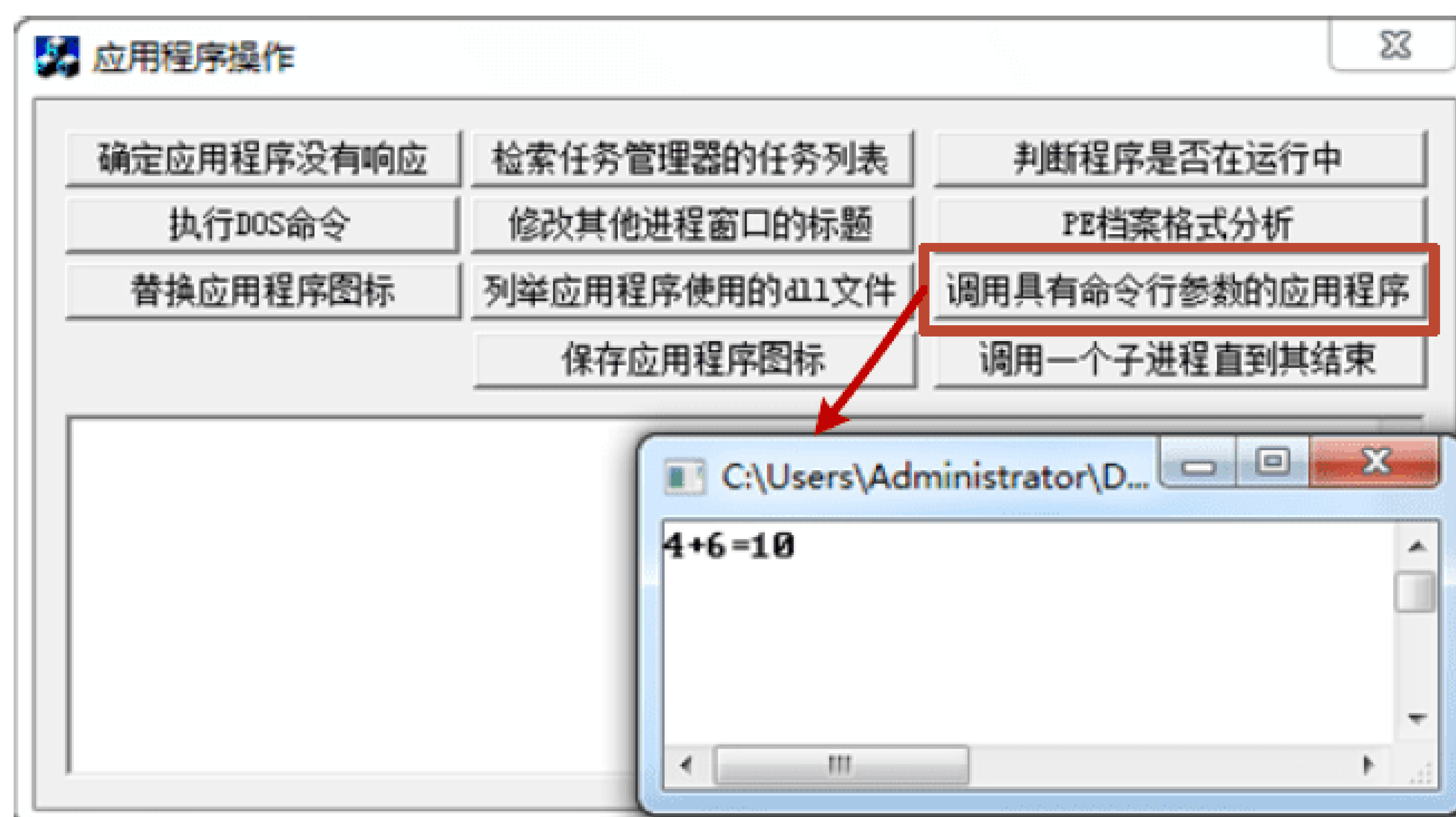


图 20-31　调用具有命令行参数的应用程序的运行效果

20.4.12　在程序中调用一个子进程直到结束

调用 CreateProcess()函数可以创建运行指定程序的进程，通过 WaitForSingleObject()等待函数可以在程序中实现等待子进程结束才返回的功能。WaitForSingleObject()函数的第一个参数使用 CreateProcess()函数返回的进程句柄，第二个参数设置为 INFINITE，表示无超时地等待，直到子进程退出才返回程序。代码如下：

```
01  //在程序中调用一个子进程直到结束
02  void CAppOperSampleDlg::OnButtonWaitprocess()
03  {
04      WriteLog("调用计算器程序......");          //显示提示信息
05      STARTUPINFO si={0};                      //定义启动信息变量
06      PROCESS_INFORMATION pi;                  //定义进程信息变量
07      si.cb = sizeof(si);                      //赋值进程结构大小
08      //启动程序
09      if(CreateProcess(NULL,"calc.exe",NULL,NULL,false,
10                      0,NULL,NULL,&si,&pi))
11      {
12          WaitForSingleObject(pi.hProcess,INFINITE);//等待程序返回
13          WriteLog("计算器程序正常结束.");           //显示结果信息
14      }
15  }
```

上面的代码会启动“计算器”程序，只有当“计算器”程序结束后，应用程序才会在日志文本框中显示结束提示信息。运行效果如图 20-32 所示。

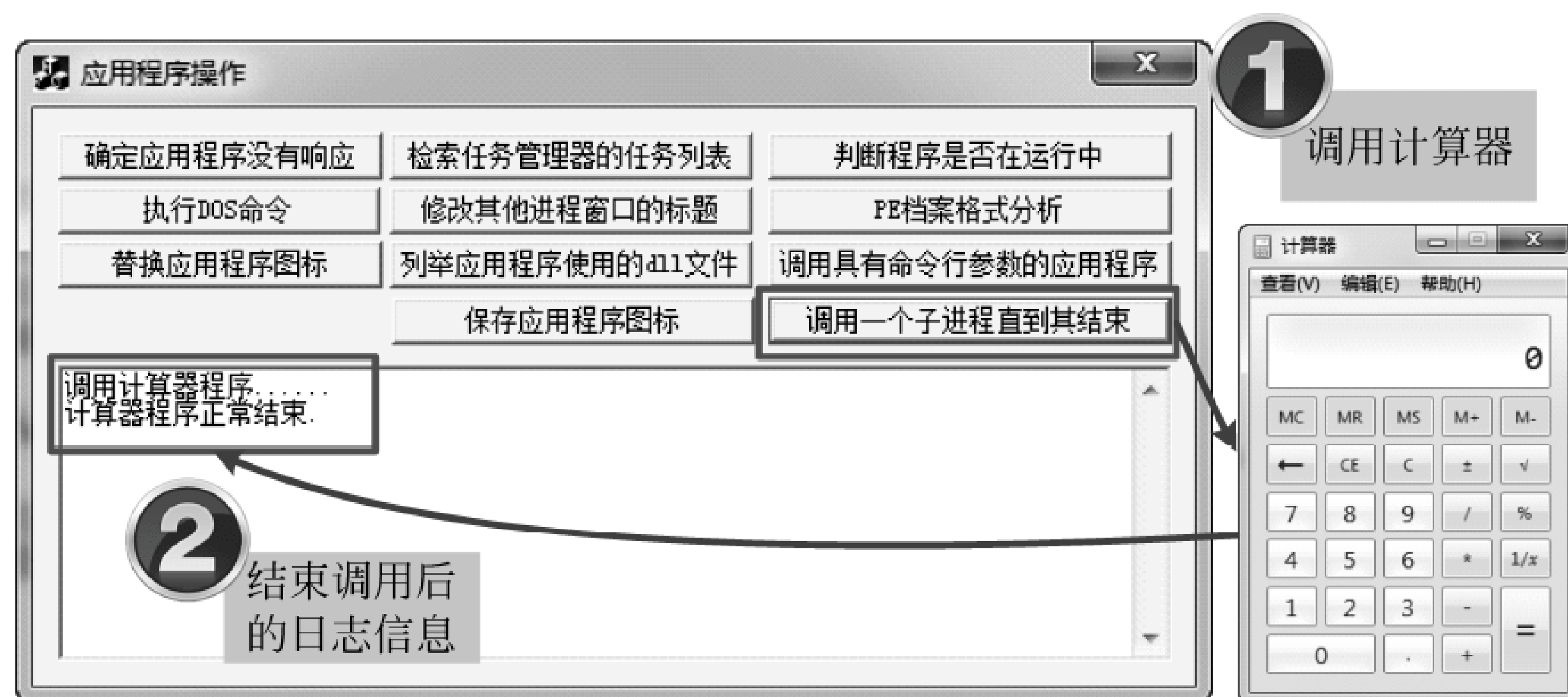


图 20-32　在程序中调用子进程直到结束运行效果

20.5　系 统 工 具

Windows 系统中提供多种系统工具，通过调用 API 或其他方式，可以直接调用这些工具，从而编写专业的实用程序。本节将介绍了如何为程序添加快捷方式、列出当前运行的进程、获取毫秒级时间、注册和卸载组件的方法、如何清空回收站以及在程序中显示文件属性对话框。

20.5.1　为程序添加快捷方式

使用 Windows Shell 可以操作 Windows 的外壳，本小节介绍如何使用 Shell 为程序添加快捷方式。IShellLink 接口是实现快捷方式的接口，IPersistFile 接口是文件接口。使用 IShellLink 接口的 IID_IPersistFile 接口方法，可以设置快捷方式对应的文件。代码如下：

```
01  void CSysToolSampleDlg::OnButtonCreatelink()//为程序添加快捷方式
02  {
03      if (!SUCCEEDED(CoInitialize(NULL)))       //初始化 COM 组件
04      {
05          WriteLog("初始化 Shell 失败");          //初始化失败显示信息
06          return;                               //返回
07      }
08      IShellLink *pisl;                         //定义快捷方式接口变量
09         //创建快捷方式实例
10      if (SUCCEEDED(CoCreateInstance(CLSID_ShellLink, NULL,
11          CLSCTX_INPROC_SERVER,IID_IShellLinkA, (void**)&pisl)))
12      {
13          IPersistFile* pIPF;                   //定义文件接口变量
14          CString szPath;                       //定义文件名变量
15          GetModuleFileName(GetModuleHandle(NULL),
16                  szPath.GetBuffer(MAX_PATH), MAX_PATH);
17           //获取文件路径
18          pisl->SetPath(szPath);         //设置文件接口的路径为当前运行路径
19          szPath.ReleaseBuffer();               //释放路径变量
```

```
20          if (SUCCEEDED(pisl->QueryInterface(IID_IPersistFile,
21                                    (void**)&pIPF)))
22          {
23              //创建文件实例
24              CString szLinkPath;                 //定义创建的快捷方式的路径
25              SHGetSpecialFolderPath(0,szLinkPath.GetBuffer(MAX_PATH),
26                  CSIDL_DESKTOPDIRECTORY, 0);     //获取桌面对应的路径
27              szLinkPath.ReleaseBuffer();         //释放快捷方式路径变量
28              szLinkPath += szPath.Mid(szPath.ReverseFind('\\'));
29              //设置快捷方式的文件名
30              szLinkPath += ".lnk";               //设置快捷方式的扩展名
31              WCHAR wpath[MAX_PATH] = { 0 };      //定义 Unicode 字符串
32              //将快捷方式的完整文件名转换为 Unicode
33              MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, szLinkPath,
34                  -1, wpath, MAX_PATH);
35              pIPF->Save(wpath, false);           //设置快捷方式对应的文件名
36              pIPF->Release();                    //释放文件接口
37              //显示创建的快捷方式信息
38              WriteLog("创建快捷方式成功，快捷方式路径=%s", szLinkPath);
39          }
40          else
41              WriteLog("创建文件接口 IID_IPersistFile 失败");
42          pisl->Release();                        //释放文件接口
43      }
44      else
45          WriteLog("创建 CLSID_ShellLink 实例失败");//显示错误信息
46      CoUninitialize();                           //释放 COM 组件工作环境
47  }
```

上面代码首先调用 CoCreateInstance()函数，初始化 CLSID_ShellLink 实例到 IShellLink 接口变量中，初始化成功后，创建 IPersistFile 接口，并设置快捷方式指向当前运行的程序路径。随后将 IShellLink 与 IPersistFile 关联起来，并将 IPersistFile 接口的快捷方式文件，即 lnk 文件的路径设置为桌面路径，文件名与实际程序的名称相同。程序运行效果如图 20-33 所示。

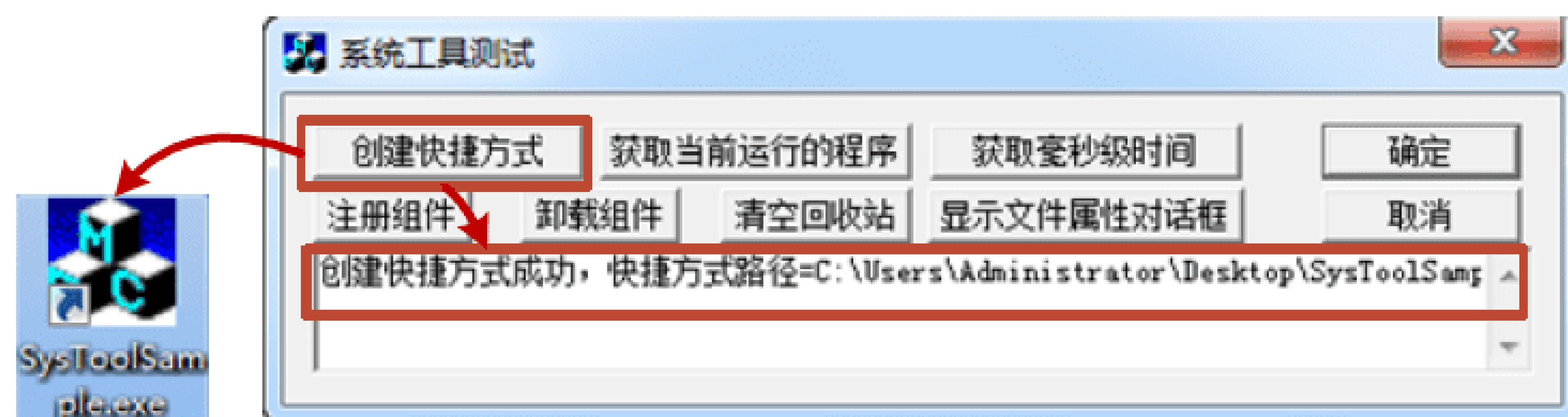


图 20-33　创建快捷方式运行效果

20.5.2　显示系统正在运行的程序

要列出系统正在运行的程序，首先需要 CreateToolhelp32Snapshot()函数取得当前系统运行的快照，然后调用 Process32First()函数启动进程枚举工作，然后依次调用 Process32Next()函数枚举下一个运行的进程。在使用 Process32First()函数和 Process32Next()函数枚举正在运行的进程时，除了要传入 CreateToolhelp32Snapshot()函数返回的快照句柄

外，还需要传入 PROCESSENTRY32 结构类型的变量，函数会将返回的信息存储在此结构中。此结构的定义为：

```
typedef struct tagPROCESSENTRY32 {
  DWORD dwSize;                          //指定结构的长度字节数
  DWORD cntUsage;                        //指定进程引用数
  DWORD th32ProcessID;                   //进程 ID 标识进程
  DWORD th32DefaultHeapID;               //指定进程的默认堆栈标识符
  DWORD th32ModuleID;                    //进程的模块标识符
  DWORD cntThreads;                      //指定进程启动的线程数
  DWORD th32ParentProcessID;             //进程父进程的进程标识符
  LONG  pcPriClassBase;                  //进程创建的线程的基本优先级
  DWORD dwFlags;                         //预留参数
  char szExeFile[MAX_PATH];              //指定进程的可执行文件的文件名
} PROCESSENTRY32;
```

使用 PROCESSENTRY32 结构，可以获取进程的基本信息。以下代码显示了如何列出当前运行的进程。

```
01  //显示系统正在运行的程序
02  void CSysToolSampleDlg::OnButtonGetproclist()
03  {
04      HANDLE hPS=NULL;                            //定义进程快照句柄
05      PROCESSENTRY32 pe32={0};                    //定义进程结构变量
06      //创建进程快照
07      hPS = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
08      if (hPS == (HANDLE)-1)
09      {
10          //显示错误信息
11          WriteLog("调用 CreateToolhelp32Snapshot()函数创建进程快照失败");
12          return;                                 //返回
13      }
14      pe32.dwSize = sizeof(PROCESSENTRY32);       //为进程结构的大小分量赋值
15      if (Process32First(hPS, &pe32))             //检索快照中的第一个进程
16      {
17          do
18          {                                       //显示检索到的进程信息
19              WriteLog("\n 优先级=%d\t 进程 ID=%d\t 线程数目=%d\t 模块名称=%s",
20                  pe32.pcPriClassBase, pe32.th32ProcessID,
21                  pe32.cntThreads,pe32.szExeFile);
22          }
23          while (Process32Next(hPS, &pe32));      //检索快照中的下一个进程
24      }
25      else
26          //显示错误信息
27          WriteLog("调用 Process32First()函数枚举运行程序失败");
28      CloseHandle (hPS);                          //关闭进程快照句柄
29  }
```

上面代码使用 CreateToolhelp32Snapshot()、Process32First()和 Process32Next()函数依次枚举了当前运行的进程，并显示出了这些进程的优先级、进程 ID、线程数目和模块名称，读者可以根据需要获取有用的数据。程序运行结果如图 20-34 所示。



图 20-34　显示系统正在运行的程序运行效果

20.5.3　如何获得毫秒级时间

使用 Windows API 函数 timeGetTime()可以获得从系统启动开始，到现在为止持续的时间，可以精确到毫秒。通过此函数可以获得毫秒级的时间。代码如下：

```
01  //如何获得毫秒级时间
02  void CSysToolSampleDlg::OnButtonGetmillsecond()
03  {
04      DWORD t1 = timeGetTime();           //获取自系统启动到现在经过的毫秒数
05      Sleep(10);                          //程序休眠 10 毫秒
06      DWORD t2 = timeGetTime();           //获取自系统启动到现在经过的毫秒数
07      WriteLog("开始时间=%u", t1);        //输出开始时间
08      WriteLog("结束时间=%u", t2);        //输出结束时间
09      //输出 10 毫秒经过的时间间隔
10      WriteLog("计时器 10 毫秒持续的时间=%u", (t2- t1));
11  }
```

上面的代码使用 timeGetTime()函数分别获得计时 10 毫秒所使用的时间，以毫秒为单位。通过运行结果可以看出，该程序实现了毫秒级的时间计数。运行效果如图 20-35 所示。

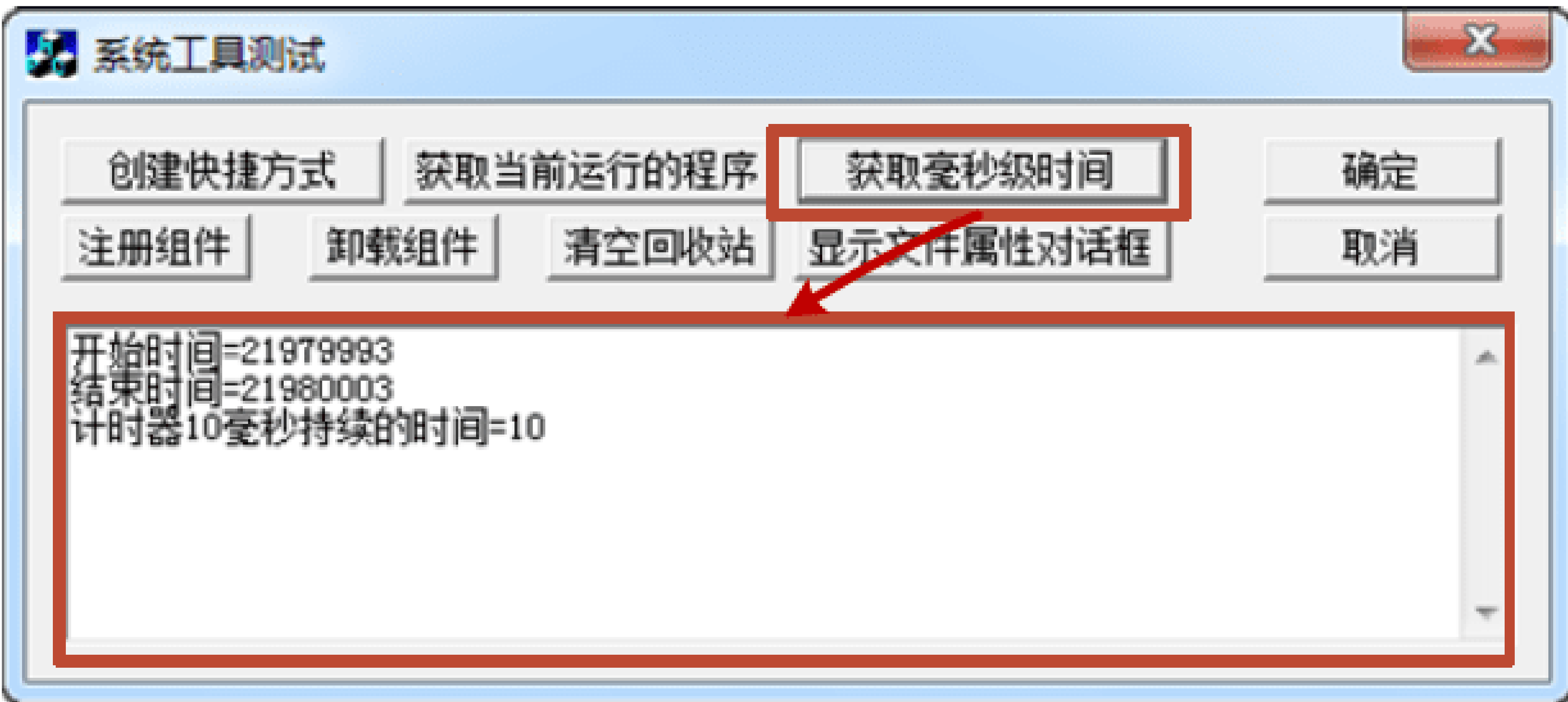


图 20-35　获得毫秒级时间运行效果

20.5.4　注册和卸载组件

在 Windows 平台下注册和卸载组件需要 3 个函数，分别是 LoadLibrary()函数、GetProcAddress()函数和 FreeLibrary()函数，作用分别是装载组件文件、获取注册或卸载函数和释放对组件文件的引用。LoadLibrary()函数返回的实例句柄在 GetProcAddress()函数和

FreeLibrary()函数中作为操作标识。下面是注册和卸载组件的代码。

```
01  void CSysToolSampleDlg::OnButtonRegistcom()      //注册组件
02  {
03      CString szPath= "COMSample.dll";             //定义组件路径名称
04      HINSTANCE hLib=LoadLibrary(szPath);          //装载组件
05      if (hLib == NULL)                            //如果装载失败
06      {
07          WriteLog("装载%s 组件文件失败!", szPath);   //输出错误信息
08          return;                                  //返回
09      }
10      //获取 DllRegisterServer()函数指针
11      FARPROC lpEntryPoint =
12                 GetProcAddress(hLib,_T("DllRegisterServer"));
13      if (lpEntryPoint != NULL)                    //如果入口函数不为 NULL
14      {
15          //调用函数注册组件
16          if(FAILED((*lpEntryPoint)()))
17             WriteLog("调用组件%s 的 DllRegisterServer()函数失败!",szPath);
18          else
19              WriteLog("注册%s 组件成功!", szPath);   //输出成功信息
20      }
21      else
22          WriteLog("没有找到组件%s 的 DllRegisterServer()入口函数,无法注册!",
23                      szPath);
24      FreeLibrary(hLib);                           //释放对组件的引用
25  }
```

上面代码是注册组件的代码，首先调用 LoadLibrary()函数装载指定的 DLL 文件，判断返回值。如果返回值为有效取值，则调用 GetProcAddress()函数获得 DllRegisterServer()注册函数的地址指针并执行。如果执行成功，则表示注册成功。注册完成后，需要调用 FreeLibrary()函数释放对组件的引用。代码如下。

```
01  void CSysToolSampleDlg::OnButtonUnregistcom()    //卸载组件
02  {
03      CString szPath= "COMSample.dll";             //定义组件路径名称
04      HINSTANCE hLib=LoadLibrary(szPath);          //装载组件
05      if (hLib == NULL)                            //如果装载失败
06      {
07          WriteLog("装载%s 组件文件失败!", szPath);   //输出错误信息
08          return;                                  //返回
09      }
10      //获取 DllUnregisterServer()函数指针
11      FARPROC lpEntryPoint =
12             GetProcAddress(hLib,_T("DllUnregisterServer"));
13      if (lpEntryPoint != NULL)                    //如果入口函数不为 NULL
14      {
15          //调用函数卸载组件
16          if(FAILED((*lpEntryPoint)()))
17              WriteLog("调用组件%s 的 DllUnregisterServer 失败!",szPath);
18          else
19              WriteLog("卸载%s 组件成功!", szPath);   //输出成功信息
20      }
21      else
22          WriteLog("没有找到组件%s 的 DllUnregisterServer()入口函数,
```

```
23                    无法卸载！",szPath);
24       //释放对组件的引用
25       FreeLibrary(hLib);
26   }
```

上面代码是卸载组件的代码，首先调用 LoadLibrary()函数装载指定的 DLL 文件，判断返回值。如果返回值为有效取值，则调用 GetProcAddress()函数获得 DllUnregisterServer()卸载函数的地址指针并执行。如果执行成功，则卸载组件成功。卸载完成后，需要调用 FreeLibrary()函数释放对组件的引用，程序运行效果如图 20-36 所示。

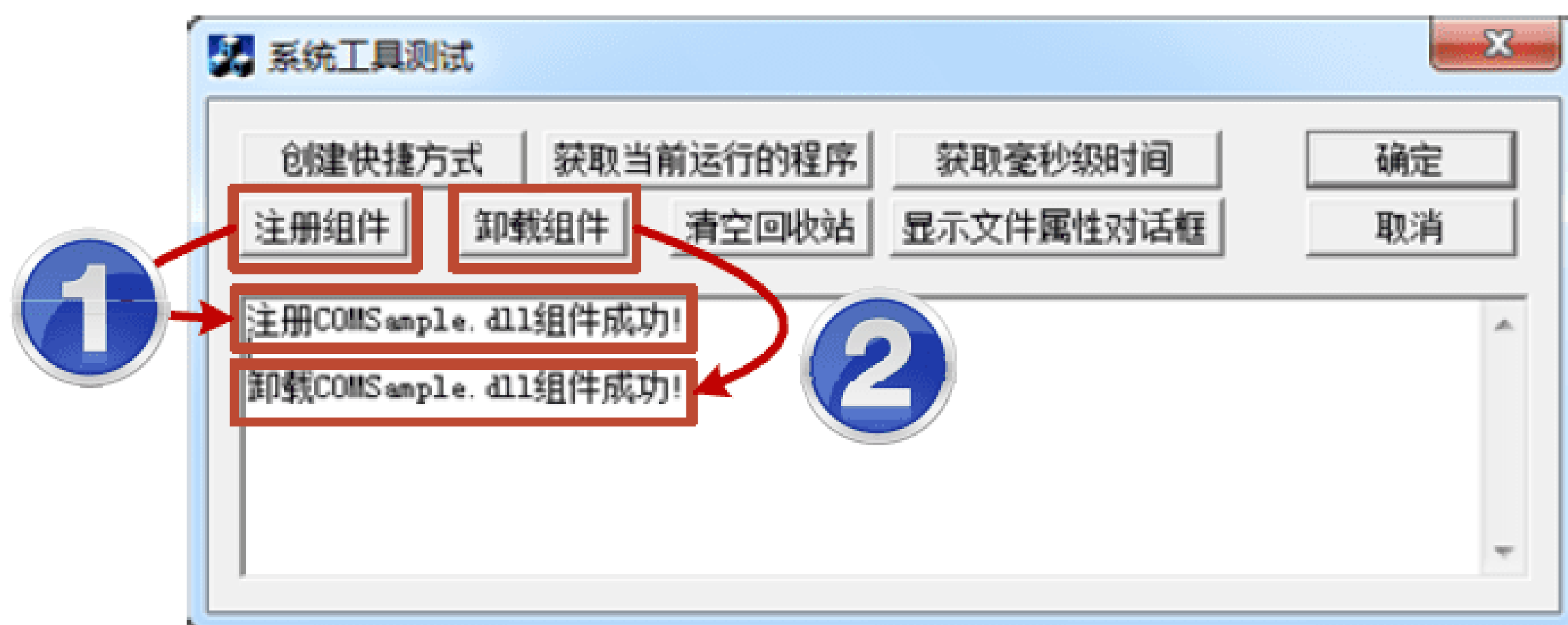


图 20-36　注册组件和卸载组件运行效果

20.5.5　清空回收站

使用 SHEmptyRecycleBin()函数可以执行清空回收站的功能。不仅可以清空指定驱动器上的回收站，还可以清空所有驱动器上的回收站，并且可以指定是否显示删除提示对话框。函数原型为：

```
SHSTDAPI SHEmptyRecycleBin(//清空回收站
  HWND hwnd,              //指定在执行清空操作时，显示的对话框的父对话框句柄
  LPCTSTR pszRootPath,    //包含要清空的回收站所在的驱动器的根目录，为 NULL 表
                          //示所有驱动器
  DWORD dwFlags);         //指定在执行清空回收站时使用的选项
```

SHEmptyRecycleBin()函数中的 dwFlags 参数指定在执行清空回收站时使用的选项，有效选项如表 20-8 所示，其可以是其中的一项或多项组合，也可以设置为 NULL。

表 20-8　清空回收站的选项

选　　项	含　　义
SHERB_NOCONFIRMATION	不显示确认删除对象的对话框
SHERB_NOPROGRESSUI	不显示清空回收站的进度
SHERB_NOSOUND	当操作完成后，不播放提示声音

下面的代码显示了清空回收站的方法。

```
01  void CSysToolSampleDlg::OnButtonClearbin()                //清空回收站
02  {
03      if( SHEmptyRecycleBin(this->m_hWnd, NULL, NULL) == S_OK)
04          //清空回收站
05          WriteLog("清空回收站完成");                       //输出成功信息
```

```
06      else
07          WriteLog("清空回收站失败");                          //输出错误信息
08   }
```

上面代码使用 SHEmptyRecycleBin()函数清空所有驱动器上的回收站，并且显示提示对话框和清空进度，并在清空完成后播放提示声音。程序运行效果如图 20-37 所示。

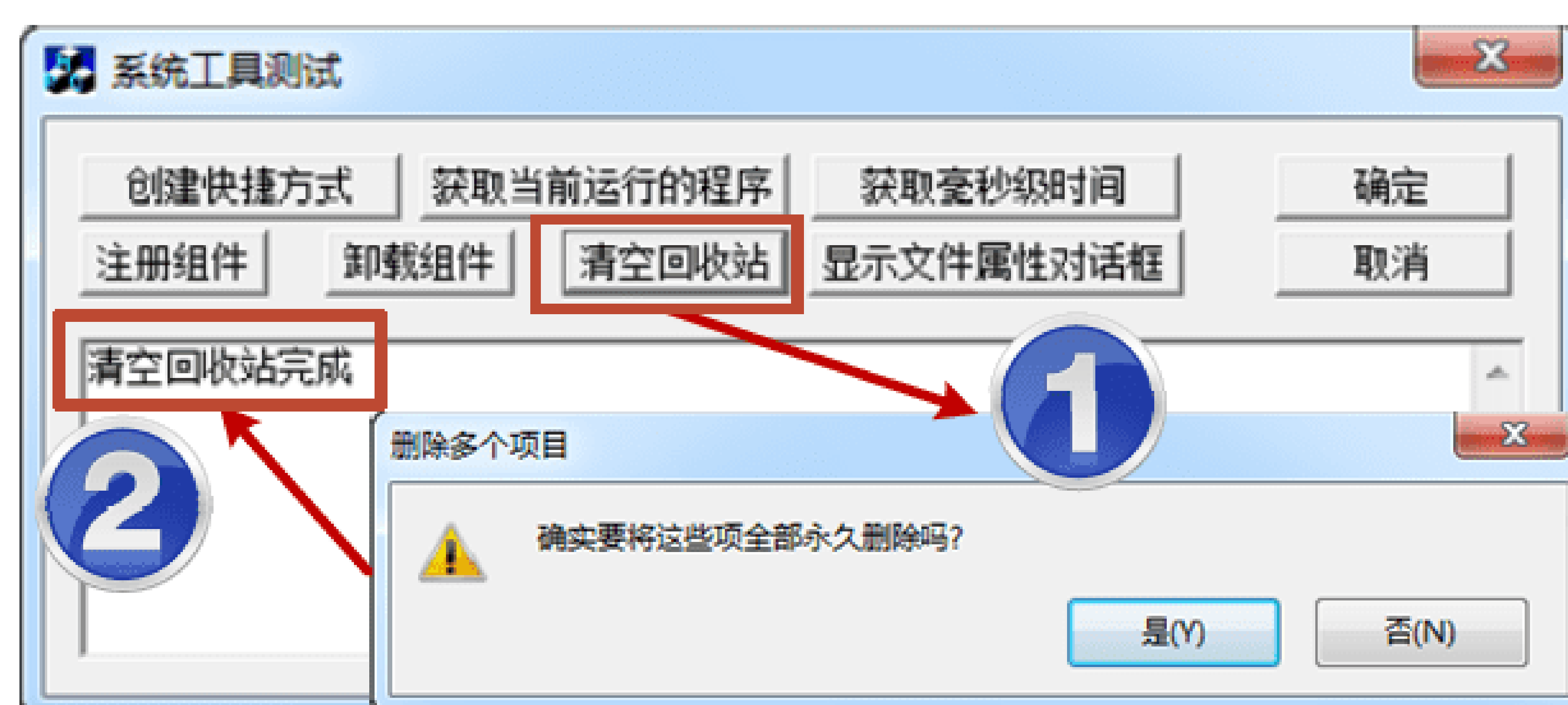


图 20-37　清空回收站运行效果

20.5.6　如何在程序中显示文件属性对话框

调用 ShellExecuteEx()函数可以完成对文件的操作。LPSHELLEXECUTEINFO 结构的参数，用于指定要进行操作的对象和内容。函数返回 BOOL 值来标识操作是否成功。LPSHELLEXECUTEINFO 结构的定义为：

```
typedef struct  SHELLEXECUTEINFO{
  DWORD cbSize;                      //指定此结构的大小
  ULONG fMask;                       //标记当前结构成员的内容有效性
  HWND hwnd;                         //指定操作执行过程中，接收消息的窗体句柄
  LPCTSTR lpVerb;                    //指定操作类型，默认是打开操作
  LPCTSTR lpFile;                    //指定要操作的文件的名称
  LPCTSTR lpParameters;              //指定运行应用程序的参数
  LPCTSTR lpDirectory;               //指定文件所在的目录
  int nShow;                         //指定显示或隐藏文件
  HINSTANCE hInstApp;                //返回操作的应用实例
  LPVOID lpIDList; LPCSTR lpClass; HKEY hkeyClass; DWORD dwHotKey;
  HANDLE hIcon; HANDLE hProcess;
} SHELLEXECUTEINFO, FAR *LPSHELLEXECUTEINFO;
```

下面的代码显示了如何调用文件的属性对话框。

```
01  //在程序中显示文件属性对话框
02  void CSysToolSampleDlg::OnButtonShowprodig()
03  {
04      CString szPath;                                  //定义路径
05      //获取当前运行路径
06      GetCurrentDirectory(MAX_PATH, szPath.GetBuffer(MAX_PATH));
07      szPath.ReleaseBuffer();                          //释放路径变量
08      SHELLEXECUTEINFO   seci;                         //定义可执行信息
09      ZeroMemory(&seci,sizeof(seci));                  //初始化可行性信息变量
10      seci.cbSize = sizeof(SHELLEXECUTEINFO);          //赋值可执行变量的大小
11      seci.hwnd = this->m_hWnd;                        //赋值窗体句柄
12      seci.lpParameters = NULL;                        //赋值参数为 NULL
```

```
13        seci.lpDirectory = szPath;                    //赋值文件夹
14        seci.nShow = 0;                               //不显示文件
15        seci.hInstApp = 0;                            //赋值应用实例
16        szPath += "\\ReadMe.txt";                     //文件名
17        seci.lpFile = szPath;                         //赋值文件名
18        seci.lpVerb = "properties";                   //赋值调用版本为属性
19        //设置选项
20        seci.fMask=SEE_MASK_NOCLOSEPROCESS|SEE_MASK_INVOKEIDLIST
21                  |SEE_MASK_FLAG_NO_UI;
22        ShellExecuteEx(&seci);                        //调用文件属性对话框
23    }
```

上面代码定义了一个 SHELLEXECUTEINFO 结构的变量，将其传入 ShellExecuteEx() 函数，在 lpDirectory 成员变量中和 lpFile 成员变量中指定要查看属性的文件。程序的运行效果如图 20-38 所示，会在程序中弹出文件的属性对话框。

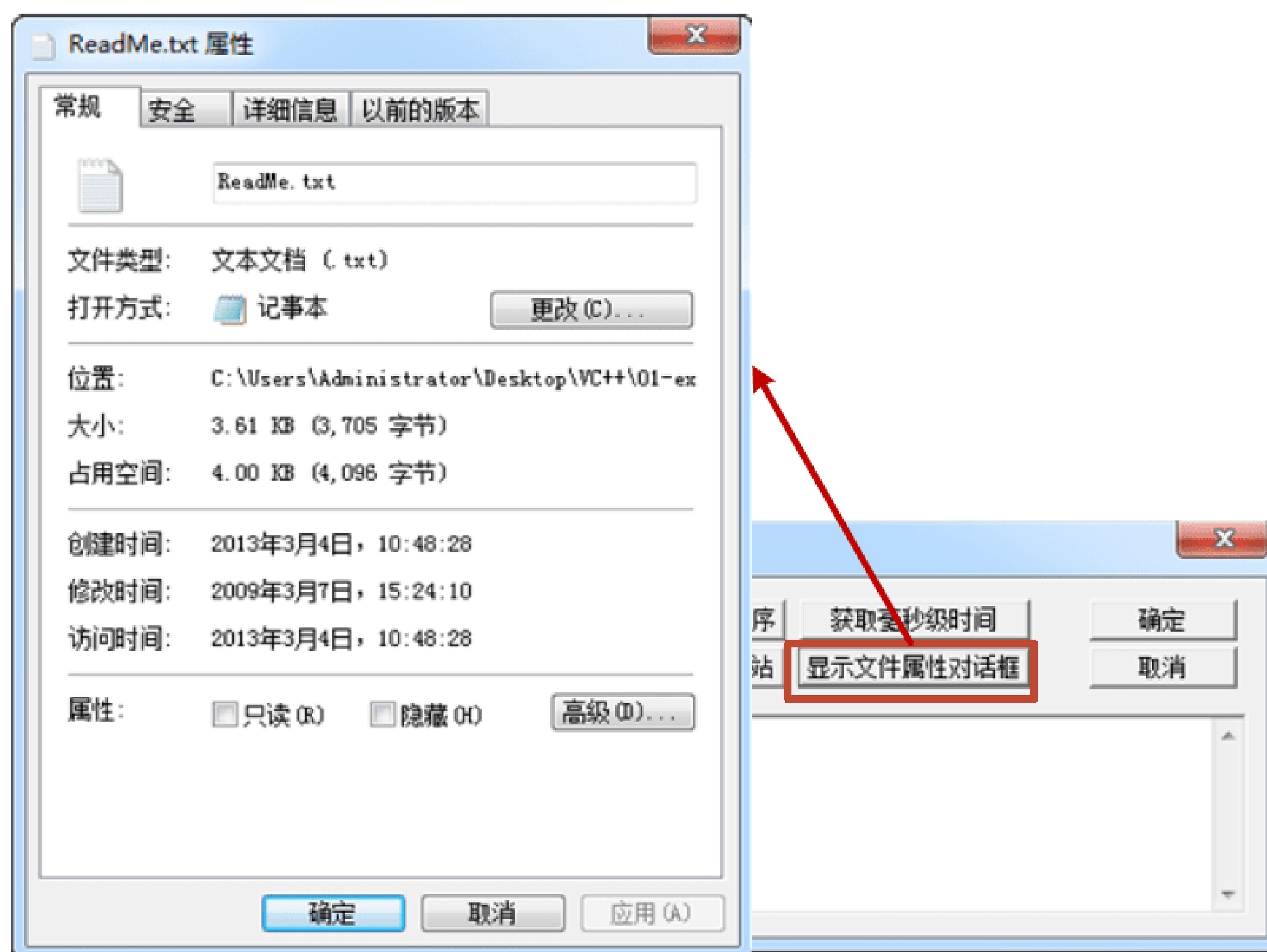


图 20-38　在程序中显示文件属性对话框运行效果

20.6　桌 面 相 关

Windows 操作系统的特点就是图形化操作系统，每个操作对象都是一个窗体，因此称为 Windows。实际上，在 Windows 操作系统中，桌面也是一个对话框。本节就介绍与桌面相关的操作，如获取桌面和任务栏对话框、显示和隐藏桌面文件、显示和隐藏 Windows 任务栏、显示和隐藏开始按钮、显示和隐藏任务栏时钟、判断屏保是否在运行、判断系统使用的字体和改变桌面背景颜色等。

20.6.1　获取桌面对话框

当启动 Windows 操作系统时，会自动创建桌面对话框。桌面对话框是系统定义的对话

框，用于绘制屏幕的背景，是所有应用程序显示的对话框的基础。桌面对话框使用位图绘制屏幕的背景，此位图称为“桌面墙纸”。默认情况下，桌面对话框使用注册表中的 BMP 文件作为桌面墙纸。

桌面对话框包括了整个屏幕，所有图标或其他对话框都在桌面对话框区域的最上层绘制。Win32 API 提供了 GetDesktopWindow()函数类来获取桌面对话框，此函数定义在 winuser.h 文件中。函数原型为：

```
HWND GetDesktopWindow(VOID)
```

此函数没有参数，并且返回桌面对话框的句柄。有关桌面的操作，必须首先获取桌面句柄，然后调用相应的函数处理桌面操作。在后面的几小节中会看到有关桌面对话框的扩展应用。

20.6.2　获取任务栏对话框句柄

在 Window API 中，使用 FindWindow()函数可以获取指定类名和对话框名的最上层对话框的句柄，此函数不会获取所有具有此类名和对话框名的子对话框。其函数原型为：

```
HWND FindWindow(
  LPCTSTR lpClassName,                    //指定对话框的类名
  LPCTSTR lpWindowName );                 //对话框标题
```

上面的函数获取类名为 lpClassName、对话框标题为 lpWindowName 的对话框。如果 lpWindowName 参数设置为 NULL，则会匹配所有具有指定类名的对话框。如果函数成功，则返回值为代表指定类名和对话框名的对话框的句柄。下面是获取任务栏对话框句柄的代码。

```
01  //获取任务栏对话框句柄
02  void CDesktopSampleDlg::OnButtonGetBar()
03  {
04      HWND hWinBar = ::FindWindow("Shell_TrayWnd","");//获取任务栏对话框
05      if(hWinBar != NULL)
06          WriteLog("获取任务栏窗口句柄成功=0x%08X", hWinBar);
07      //输出信息
08      else
09          WriteLog("获取任务栏窗口句柄失败");                //输出错误信息
10  }
```

上面代码使用 FindWindow()函数获取任务栏对话框句柄，其中 Shell_TrayWnd 参数表示任务栏对话框的类名。通过判断返回值是否为 NULL，判断是否成功获取对话框句柄，同时将操作结果显示在日志文本框中。程序运行效果如图 20-39 所示。

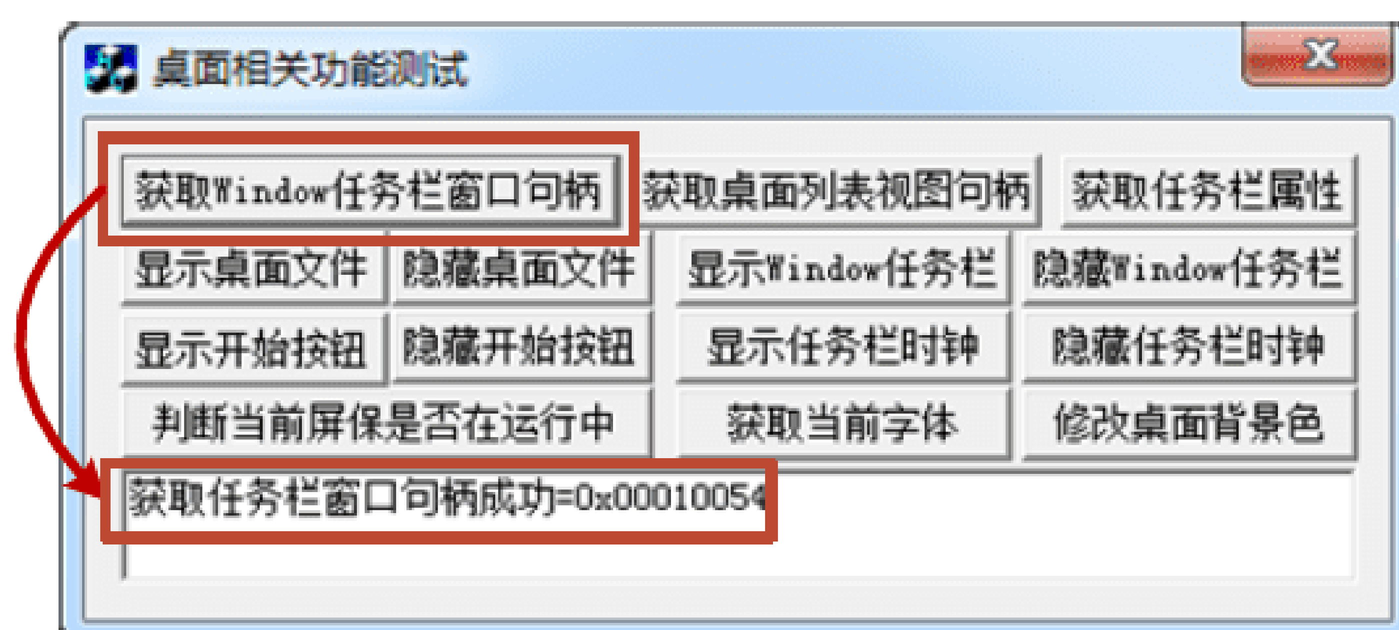


图 20-39　获取任务栏句柄运行效果

20.6.3　获取桌面列表视图句柄

获取桌面句柄后，可以获取桌面上的对象，如桌面上的文件列表就是桌面上的列表视图对象，它是桌面窗体对象的对象。具体代码如下：

```
01  //获取桌面列表视图句柄
02  void CDesktopSampleDlg::OnButtonGetDesklist()
03  {
04      HWND hDeskWnd = ::FindWindow("Progman",NULL);//获取桌面程序对象
05      if(hDeskWnd == NULL)                        //如果获取失败
06      {
07          WriteLog("获取桌面句柄失败。");
08          return;      //显示错误信息并返回
09      }
10      HWND  hSubDeskWnd=::GetDlgItem(hDeskWnd, 0L);
11      //获取桌面对象中的第一个对象
12      if(hSubDeskWnd == NULL)                     //如果获取失败
13      {
14          WriteLog("获取桌面窗体中的对象失败。");
15          return;  //显示错误信息并返回
16      }
17      HWND hDeskList=::GetDlgItem(hSubDeskWnd, 1L);
18      //获取桌面对象的第一个对象中的第二个对象
19      if(hDeskList == NULL)                       //如果获取失败
20      {
21         //显示错误信息并返回
22          WriteLog("获取桌面窗体中的对象失败。");
23          return;
24      }
25      //输出获取的信息
26      WriteLog("获取桌面列表视图句柄成功=0x%08X",hDeskList);
27  }
```

上面代码首先获取桌面句柄，然后获取桌面上的第一个对象中的第二个对象，其中的第二个对象就是桌面列表视图句柄。程序运行效果如图 20-40 所示。

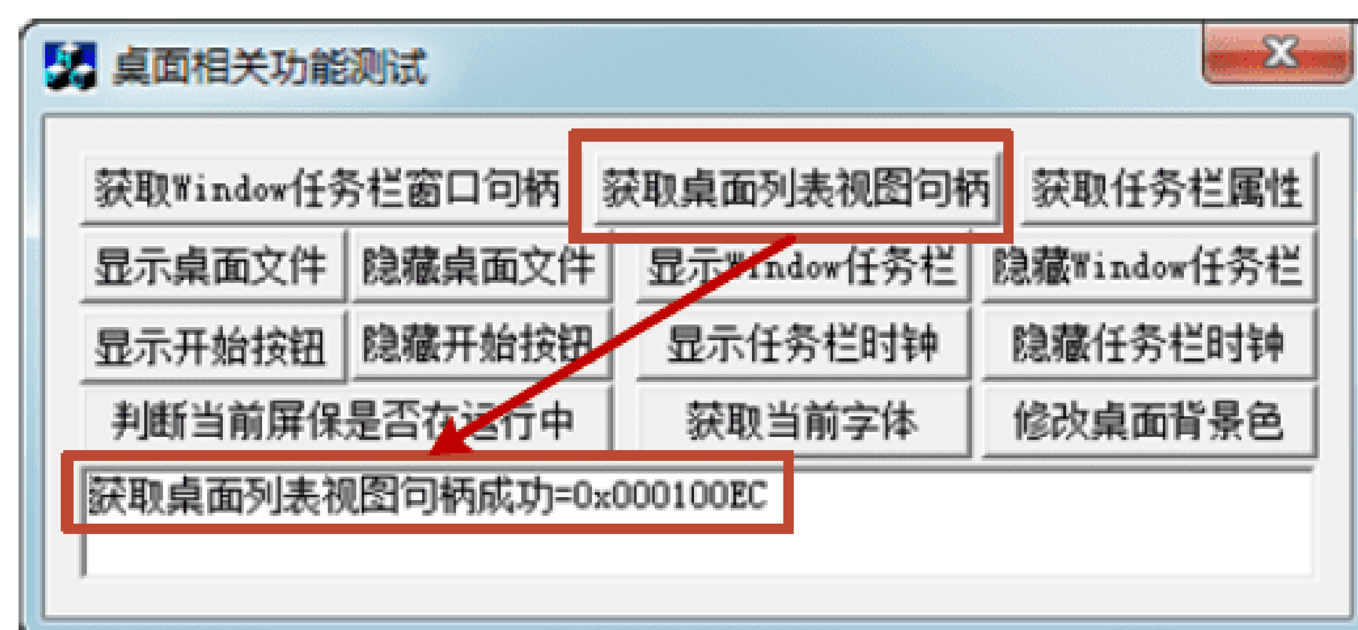


图 20-40　获取桌面列表视图句柄运行效果

20.6.4　获取任务栏属性

在 Windows 程序中，调用 Shell 内核可以控制 Windows 外壳操作，因此，本小节使用这种方式获取任务栏属性。调用 CoCreateInstance()函数创建 Shell 实例，并将获得的 Shell

句柄存入变量，通过调用此变量的 TrayProperties()方法弹出任务栏的属性对话框。代码如下：

```
01  void CDesktopSampleDlg::OnButtonGettoolbarpro()//获取任务栏属性
02  {
03      if(FAILED(CoInitialize(NULL)))              //初始化 COM 工作环境
04      {
05          WriteLog("初始化 COM 工作环境失败");
06          return;  //显示错误信息并返回
07      }
08      IShellDispatch* pShellDispatch=NULL;//定义 IShellDispatch 接口变量
09      //创建 IShellDispatch 实例
10      if(SUCCEEDED(CoCreateInstance(CLSID_Shell,NULL,
11         CLSCTX_INPROC_SERVER, IID_IDispatch,
12         (LPVOID*)&pShellDispatch)))
13      {
14          if(SUCCEEDED(pShellDispatch->TrayProperties()))//显示任务栏
15              WriteLog("显示任务栏属性窗口成功");          //显示成功信息
16          else
17              WriteLog("显示任务栏属性窗口失败");          //显示错误信息
18      }
19      else
20          WriteLog("创建 IShellDispatch 接口实例失败");    //显示错误信息
21      CoUninitialize();                                //释放 COM 工作环境
22  }
```

上面代码调用 CoCreateInstance()函数创建 IShellDispatch 实例对象，并调用 TrayProperties()方法显示任务栏窗口，程序的运行效果如图 20-41 所示。

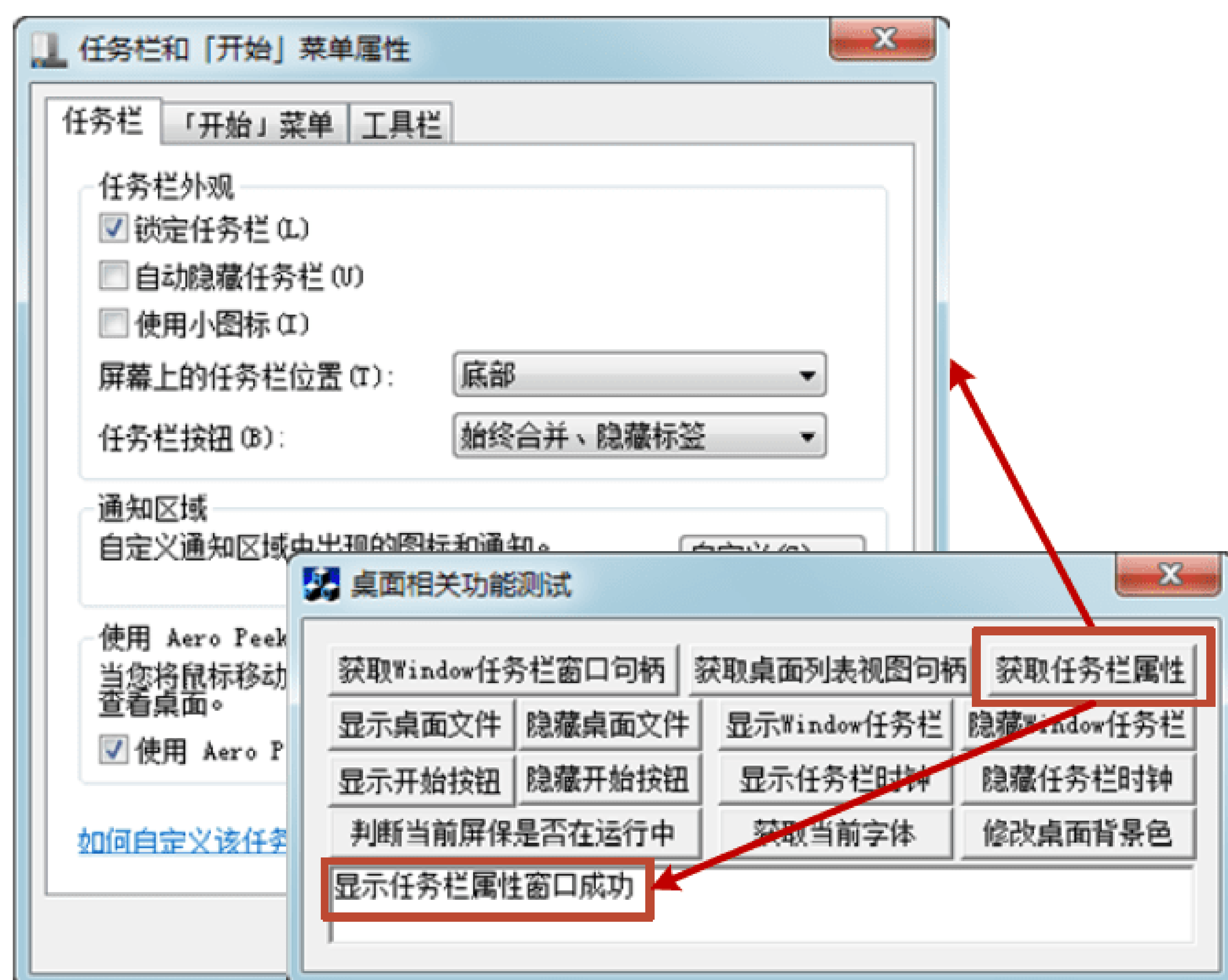


图 20-41　获取任务栏属性的程序运行效果

20.6.5　隐藏和显示桌面图标

在 Windows 系统中，桌面实际上是一个对话框，只不过是一个列表对话框。要控制桌面图标的显示与否，首先需要获取代表桌面文件列表的句柄，然后控制此句柄代表的窗体

是否显示。代码如下：

```
01  void CDesktopSampleDlg::OnButtonHideDesktop()       //隐藏桌面图标
02  {
03      HWND hDeskWnd = ::FindWindow("Progman",NULL);   //获取桌面窗体句柄
04      if(hDeskWnd!=NULL)
05          ::ShowWindow(hDeskWnd,SW_HIDE);             //隐藏桌面图标
06  }
07  void CDesktopSampleDlg::OnButtonShowDesktop()       //显示桌面图标
08  {
09      HWND hDeskWnd = ::FindWindow("Progman",NULL);   //获取桌面窗体句柄
10      if(hDeskWnd!=NULL)
11          ::ShowWindow(hDeskWnd,SW_SHOW);             //显示桌面图标
12  }
```

上面代码使用 FindWindow()函数获取桌面文件列表句柄，Progman 参数表示桌面对话框的类名。通过判断返回值是否为 NULL，判断是否成功获取对话框句柄。如果获取了桌面对话框，则调用 ShowWindow()函数控制其显示与否，SW_SHOW 表示显示，SW_HIDE 表示隐藏。程序运行效果如图 20-42 所示。

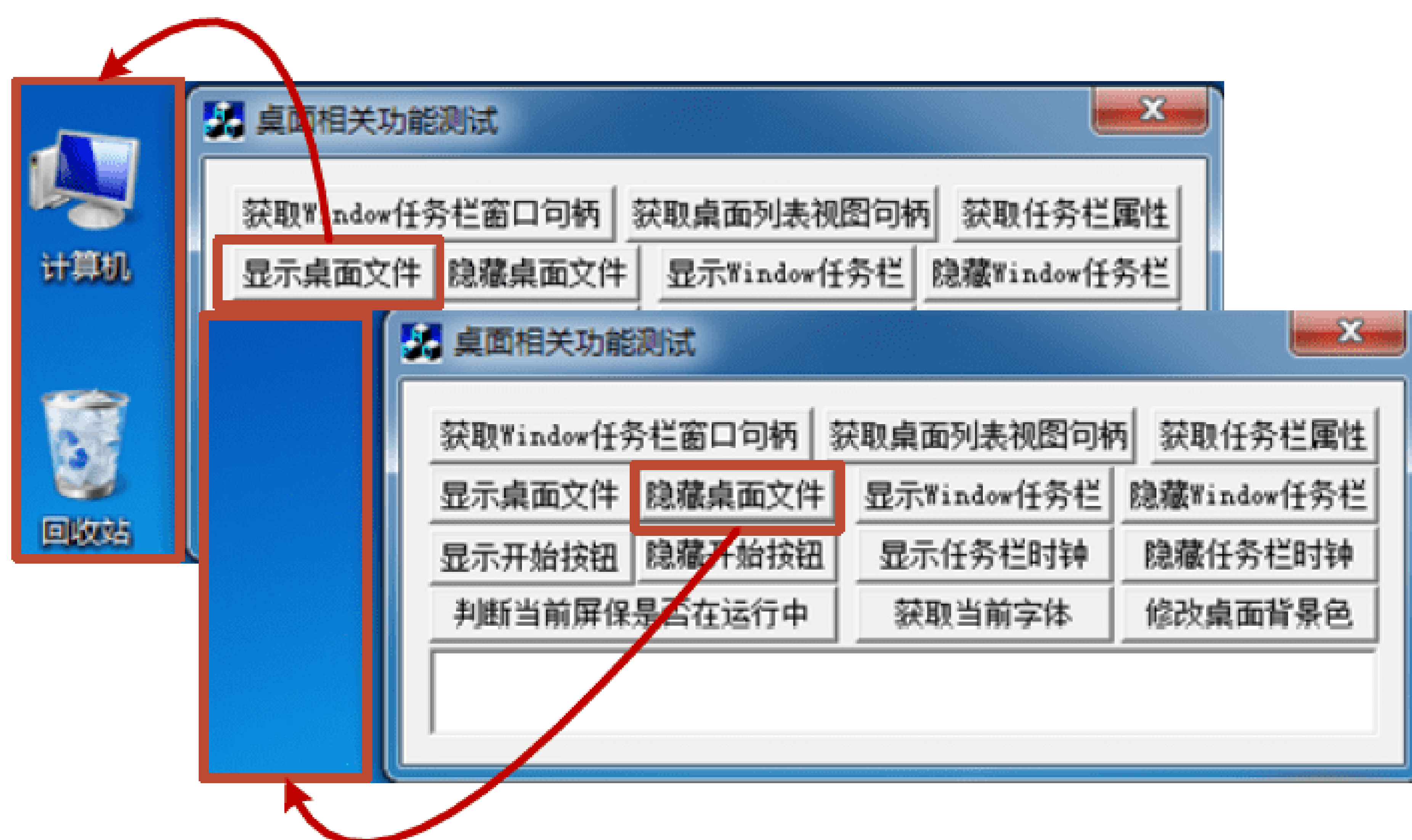


图 20-42　隐藏和显示桌面图标运行效果

20.6.6　隐藏和显示 Windows 任务栏

在 Windows 系统中，任务栏实际上是一个子对话框，要控制任务栏的显示与否，首先需要获取任务栏的句柄，然后控制此句柄代表的窗体是否显示。代码如下：

```
01  void CDesktopSampleDlg::OnButtonHideBar()           //隐藏任务栏
02  {
03      HWND hWinBar = ::FindWindow("Shell_TrayWnd","");//获取任务栏句柄
04      if(hWinBar!=NULL)
05          ::ShowWindow(hWinBar, SW_HIDE);             //隐藏任务栏
06  }
07  void CDesktopSampleDlg::OnButtonShowBar()           //显示任务栏
08  {
09      HWND hWinBar = ::FindWindow("Shell_TrayWnd","");//获取任务栏句柄
10      if(hWinBar!=NULL)
```

```
11          ::ShowWindow(hWinBar, SW_SHOW);//显示任务栏
12   }
```

其中，OnButtonShowBar()函数是显示任务栏的处理函数，使用 FindWindow()函数传入 Shell_TrayWnd 参数获取 Windows 任务栏对话框，使用 ShowWindow()函数显示 Windows 任务栏。隐藏 Windows 任务栏的过程与显示任务栏的处理是相似的，只是最后使用 ShowWindow()函数的 SW_HIDE 隐藏 Windows 任务栏。程序运行效果如图 20-43 所示。

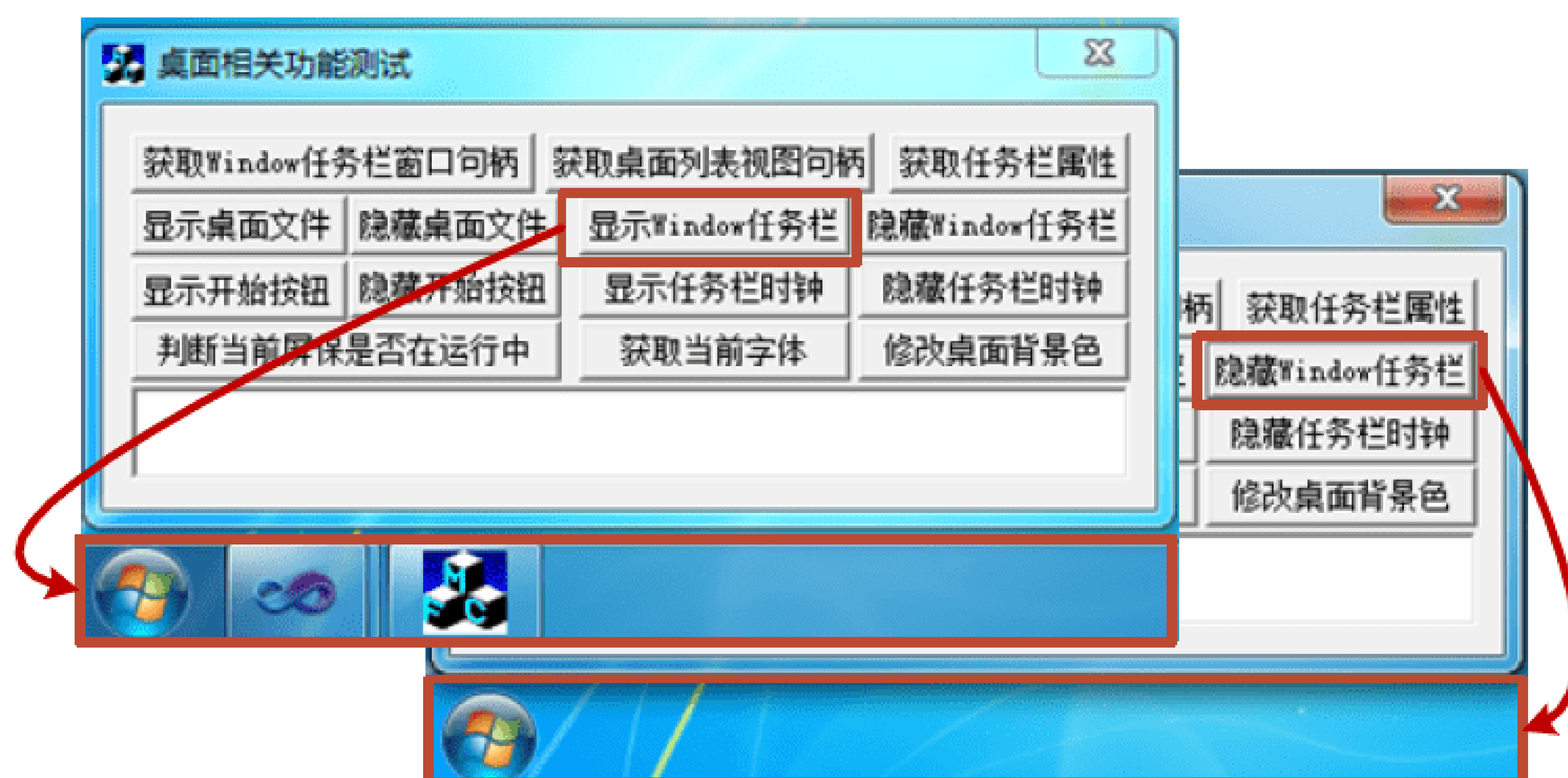


图 20-43　隐藏和显示 Windows 任务栏运行效果

20.6.7　隐藏和显示“开始”按钮

在 Windows XP 中，“开始”按钮实际上是一个子对话框，要控制“开始”按钮的显示与否，首先需要获取“开始”按钮的句柄，然后控制此句柄代表的对话框是否显示。Windows 中使用 Shell_TrayWnd 表示 Windows 任务栏，其中的 Button 表示“开始”按钮。代码如下：

```
01   void CDesktopSampleDlg::OnButtonShowStart()          //显示“开始”按钮
02   {
03       HWND hWinBar = ::FindWindow("Shell_TrayWnd","");//获取任务栏句柄
04       if(hWinBar!=NULL)
05       {
06           HWND hMenu = ::FindWindowEx(hWinBar, 0, "Button", NULL);
07           //获取“开始”按钮句柄
08           if(hMenu!=NULL)
09               ::ShowWindow(hMenu, SW_SHOW);            //显示“开始”按钮
10           else
11               WriteLog("获取开始按钮对话框句柄失败");      //输出错误信息
12       }
13       else
14           WriteLog("获取 Windows 任务栏句柄失败");        //输出错误信息
15   }
16   void CDesktopSampleDlg::OnButtonHideStart()          //隐藏“开始”按钮
17   {
18       HWND hWinBar = ::FindWindow("Shell_TrayWnd","");//获取任务栏句柄
19       if(hWinBar!=NULL)
20       {
21           HWND hMenu = ::FindWindowEx(hWinBar, 0, "Button", NULL);
22           //获取“开始”按钮句柄
23           if(hMenu!=NULL)
```

```
24              ::ShowWindow(hMenu, SW HIDE);          //隐藏“开始”按钮
25          else
26              WriteLog("获取开始按钮对话框句柄失败");    //输出错误信息
27      }
28      else
29          WriteLog("获取 Windows 任务栏句柄失败");    //输出错误信息
30  }
```

其中，OnButtonShowStart()函数是显示“开始”按钮的处理函数，调用 FindWindow()函数传入 Shell_TrayWnd 参数获取 Windows 任务栏对话框，然后调用 FindWindowEx()函数获取 Windows 任务栏中的“开始”按钮句柄，最后，调用 ShowWindow()函数显示“开始”按钮。隐藏“开始”按钮的过程与显示“开始”按钮的过程是相同的，只是最后需要调用 ShowWindow()函数隐藏“开始”按钮。

注意：在 Windows 7 中，“开始”按钮不再是任务栏的子窗口了，而是与任务栏同级的窗口，隐藏“开始”按钮需要修改注册表。在 Windows XP 下程序运行效果如图 20-44 所示。



图 20-44　隐藏和显示“开始”按钮运行效果

20.6.8　隐藏和显示任务栏时钟

要获取 Windows 任务栏时钟，需要首先通过 FindWindow()函数，传入 Shell_TrayWnd 参数获取 Windows 任务栏句柄。通过这个句柄和 FindWindowEx()函数，传入 TrayNotifyWnd 参数获取通知区域的句柄，再通过此句柄和 FindWindowEx()函数，传入 TrayClockWClass 参数获取任务栏时钟句柄。最后，通过 ShowWindow()函数来设定要隐藏还是显示任务栏时钟。代码如下：

```
01  //隐藏任务栏时钟
02  void CDesktopSampleDlg::OnButtonHideClock()
03  {
04      HWND hWinBar = ::FindWindow("Shell TrayWnd",NULL);
05      if (hWinBar != NULL)
06      {
07          //获取通知托盘句柄
08          HWND hNotifyWnd = ::FindWindowEx(hWinBar, 0,
09                                  "TrayNotifyWnd",NULL);
10          if (hNotifyWnd != NULL)
11          {
12              //获取时钟句柄
13              HWND hClockWnd = ::FindWindowEx(hNotifyWnd, 0,
14                                  "TrayClockWClass", NULL);
15              if(hClockWnd!=NULL)
```

```
16                ::ShowWindow(hClockWnd, SW HIDE);//如果成功，则隐藏时钟
17            else
18                WriteLog("获取时钟句柄失败");         //显示错误信息
19        }
20        else
21            WriteLog("获取通知托盘句柄失败");         //显示错误信息
22    }
23    else
24        WriteLog("获取任务栏句柄失败");               //显示错误信息
25 }
26 void CDesktopSampleDlg::OnButtonShowClock()    //显示任务栏时钟
27 {
28    HWND hWinBar = ::FindWindow("Shell TrayWnd",NULL);
29    //获取任务栏句柄
30    If (hWinBar != NULL)                          //如果成功
31    {
32        //获取通知托盘句柄
33        HWND hNotifyWnd = ::FindWindowEx(hWinBar, 0,
34                                "TrayNotifyWnd",NULL);
35        if (hNotifyWnd != NULL)                   //如果成功
36        {
37            //获取时钟句柄
38            HWND hClockWnd = ::FindWindowEx(hNotifyWnd, 0,
39                               "TrayClockWClass", NULL);
40            if(hClockWnd!=NULL)
41                ::ShowWindow(hClockWnd, SW SHOW);//如果成功，则显示时钟
42            else
43                WriteLog("获取时钟句柄失败");         //显示错误信息
44        }
45        else
46            WriteLog("获取通知托盘句柄失败");         //显示错误信息
47    }
48    else
49        WriteLog("获取任务栏句柄失败");               //显示错误信息
50 }
```

在上面代码中，OnButtonHideClock()函数和 OnButtonShowClock()函数分别实现隐藏任务栏时钟和显示任务栏时钟的工作。程序运行效果如图 20-45 所示。

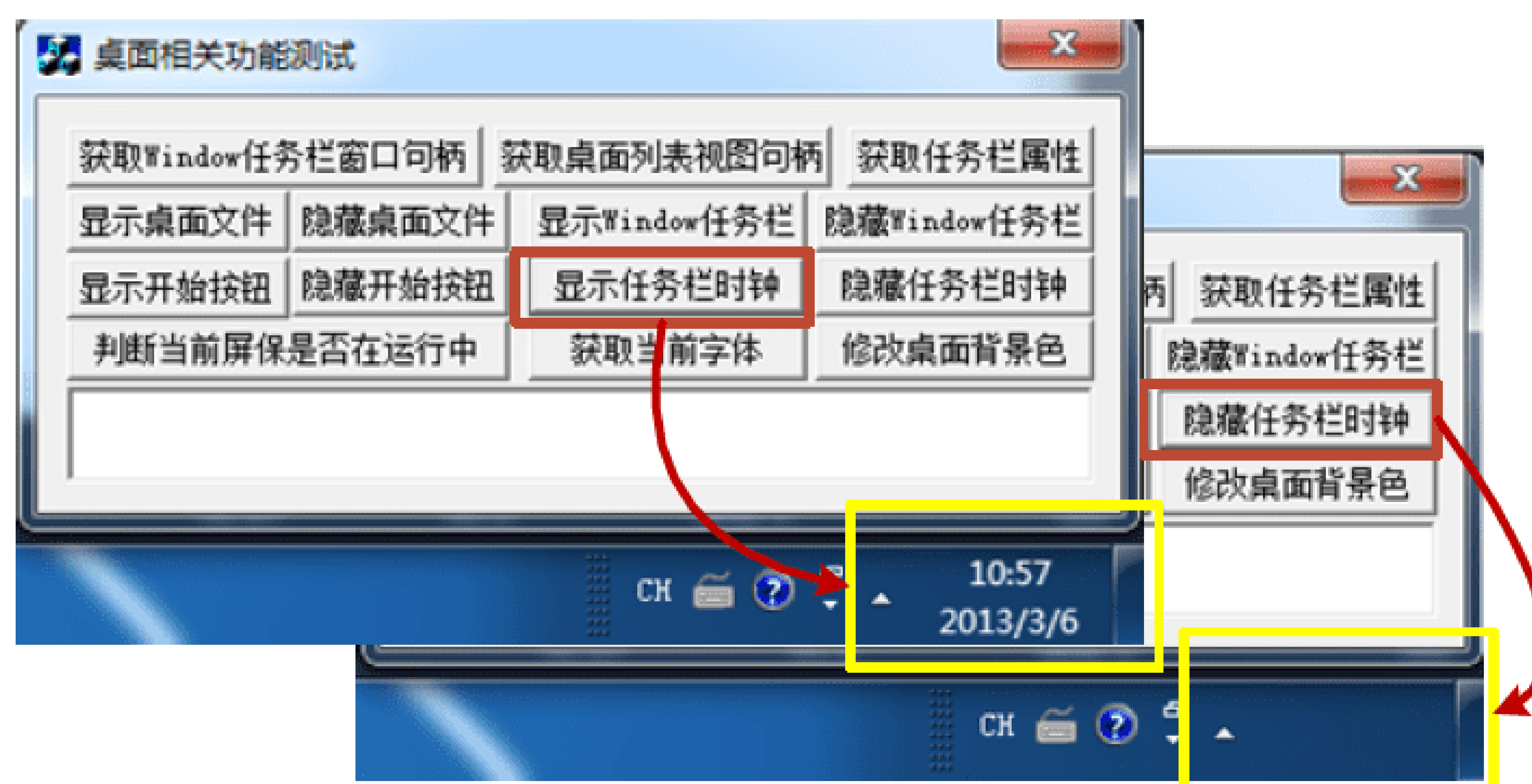


图 20-45　隐藏和显示时钟运行效果

20.6.9　判断屏幕保护程序是否在运行

通过 OpenDesktop()函数可以判断当前是否正在运行屏幕保护程序。此参数中，使用参

数名 screen-saver 标识桌面屏幕保护程序，如果使用此函数返回有效句柄，则说明屏保程序正在运行，如果没有返回有效句柄，但是其返回值为 ERROR_ACCESS_DENIED 时，则说明屏保程序正在运行。具体代码如下：

```
01  //判断屏幕保护程序是否在运行
02  void CDesktopSampleDlg::OnButtonSaverscreenRunning()
03  {
04      //打开桌面屏保程序
05      HDESK hDesktop = OpenDesktop(TEXT("screen-saver"),
06                              0, false,MAXIMUM_ALLOWED );
07      if(hDesktop == NULL)                        //如果返回的句柄无效
08      {
09          //如果返回 ERROR_ACCESS_DENIED 错误=屏保正在运行
10          if(GetLastError() == ERROR_ACCESS_DENIED)
11              WriteLog("屏保正在运行");            //输出信息提示
12          else
13              WriteLog("没有运行屏保");            //否则，屏保没有运行
14      }
15      else                                    //如果返回的句柄有效，则屏保正在运行
16      {
17          WriteLog("屏保正在运行");                   //输出信息提示
18          CloseDesktop(hDesktop);                    //关闭桌面
19      }
20  }
```

上面代码中，调用 OpenDesktop()函数打开桌面的屏幕保护程序。如果返回的错误值为 ERROR_ACCESS_DENIED 或返回有效句柄，则表示屏幕保护正在运行，否则屏幕保护没有运行。程序运行效果如图 20-46 所示。

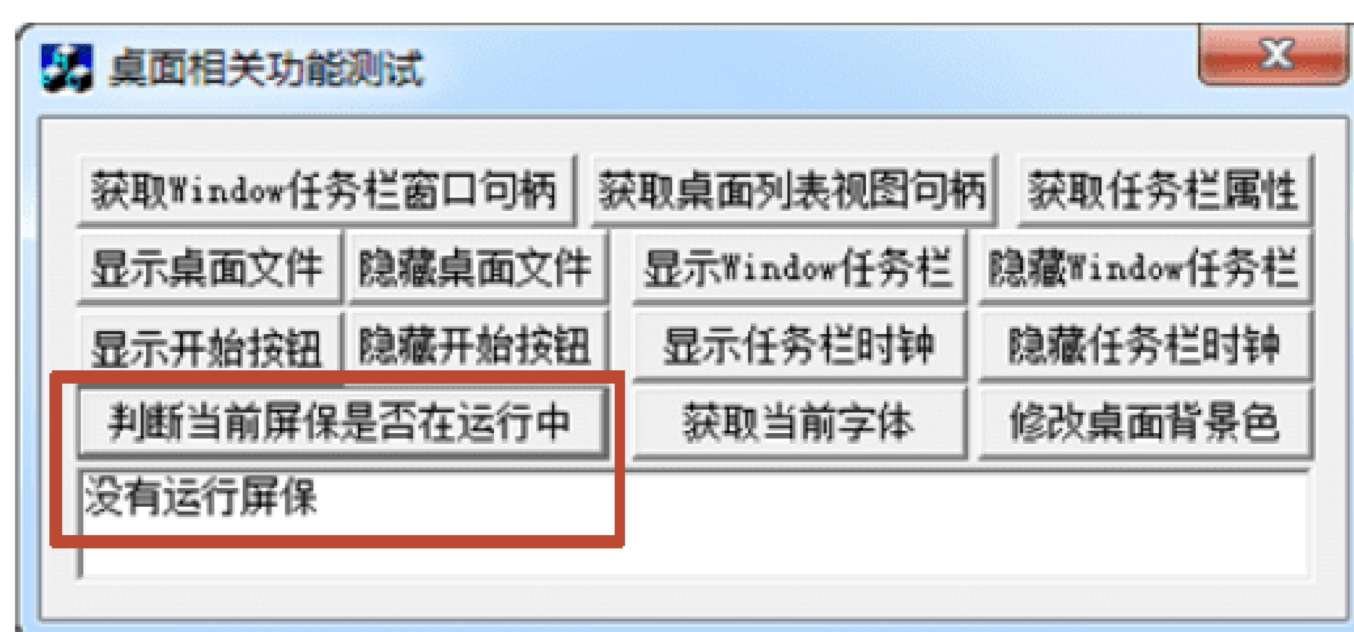


图 20-46　判断屏幕保护程序是否在运行运行效果

20.6.10　判断系统是否使用大字体

调用 Windows API 函数可以判断系统当前使用的字体。首先调用 GetDesktopWindow()函数获取桌面窗体，调用 GetWindowDC 函数获取桌面窗体对应的设备上下文，调用 GetTextMetrics()函数获取指定设备上下文使用的字体信息。在获取字体信息前，首先备份原来的设置，获取字体后，再恢复原来的模式，并释放设备上下文。最后，将获取的字体信息显示出来，并判断高度是否大于 16，如果大于 16，则表示系统使用的是大字体。代码如下：

```
01  void CDesktopSampleDlg::OnButtonGetFont()          //判断系统是否使用大字体
02  {
```

```
03      HWND hWnd = ::GetDesktopWindow();              //获取桌面句柄
04      if (hWnd != NULL)                              //如果成功
05      {
06          HDC  hDC  = ::GetWindowDC(hWnd);           //获取桌面上下文
07          if (hDC != NULL)                           //如果成功
08          {
09              //设置上下文映射模式
10              int  iOldMode = SetMapMode(hDC, MM TEXT);
11              TEXTMETRIC  tm;                    //定义字体结构变量
12              if (GetTextMetrics(hDC,&tm))       //获取桌面上下文字体信息
13              {
14                  SetMapMode(hDC,iOldMode);      //恢复桌面上下文的映射模式
15                  ::ReleaseDC(hWnd,hDC);         //释放上下文资源
16                  if (tm.tmHeight > 16)          //使用大字体
17                      WriteLog("系统使用的是大字体, 大小=%d", tm.tmHeight);
18                  else                           //没有使用大字体
19                      WriteLog("系统使用的不是大字体, 大小=%d", tm.tmHeight);
20              }
21              else
22                  WriteLog("获取字体信息失败");   //输出错误信息
23          }
24          else
25              WriteLog("获取桌面上下文失败");     //输出错误信息
26      }
27      else
28          WriteLog("获取桌面句柄失败");           //输出错误信息
29  }
```

上面代码通过 GetTextMetrics()函数获取了系统使用的字体，并判断高度是否大于 16，从而确定是否使用的是大字体。程序运行效果如图 20-47 所示。

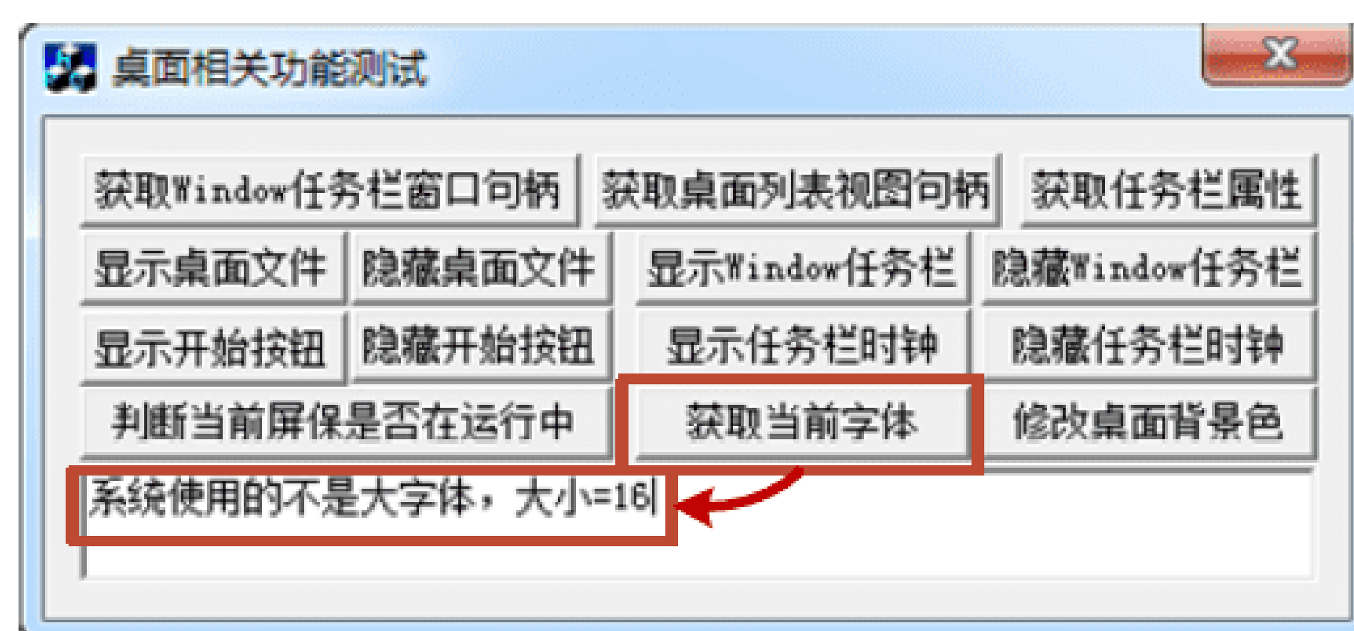


图 20-47　判断系统是否使用大字体的运行效果

20.6.11　改变桌面背景颜色

SystemParametersInfo()函数可以执行各种系统参数的设置，比如改变桌面背景的颜色。要改变桌面背景颜色，需要使用 SPI_SETDESKWALLPAPER 参数，在第三个参数指定要设置为背景的 bmp 图片路径。函数原型如下：

```
BOOL SystemParametersInfo(
  UINT uiAction,              //指定要进行的操作
  UINT uiParam,               //指定操作的第一个参数
  PVOID pvParam,              //指定操作的第二个参数，指定要设置为桌面背景的 BMP 图片
  UINT fWinIni );             //是否更新用户配置文件
```

此函数的第一个参数决定了后面两个参数的含义，在设置桌面墙纸的应用中，第二个

参数无效，使用第三个参数指定要设置的背景图片路径。下面显示了修改墙纸的代码。

```
01  void CDesktopSampleDlg::OnButtonSetBackgroud()      //改变桌面背景颜色
02  {
03      //修改桌面背景
04      if (SystemParametersInfo(SPI_SETDESKWALLPAPER, 0,
05                       "River Sumida.jpg",0))
06          WriteLog("设置桌面背景成功");                 //输出提示信息
07      else
08          WriteLog("设置桌面背景失败");                 //输出错误信息
09  }
```

上面代码将桌面墙纸设置为 River Sumida.bmp 文件。程序运行效果如图 20-48 所示。



图 20-48　设置桌面墙纸运行效果

20.7　系统信息

使用 VC 可以通过 Windows API 和注册表等方式获取有关的系统信息。读者可以通过这些系统信息编写适当的处理。包括获取有关 CPU 的信息、路径信息、操作系统信息、处理器信息、当前用户、计算机名称和当前屏幕的参数等。

20.7.1　获取 CPU ID 值

在 Windows 平台下，通过执行汇编语句可以获取 CPU ID 的值和 CPU 公司名称。使用 EAX、EBX、ECX 和 EDX 这 4 个寄存器标记，可以获取这两个值。其中，eax 中存放功能号，0 表示存放 CPU 公司，1 表示存放 CPU ID 值。Cpuid 指令表示将 CPU 信息放入寄存器中。其中，如果功能号为 0，即获取 CPU 公司值时，ebx、edx 和 ecx 中分别存放公司的前 4 个、中间 4 个和后 4 个字符。如果功能号为 1，即获取 CPU ID 时，edx 中存放 CPU ID 值。具体代码如下：

```
01  void CSystemInfoSampleDlg::OnButtonGetCpuid()//获取 CPU ID 值
02  {
03      BYTE szCPU[16]= {0};                    //定义存放 CPU 类型的数组
04      UINT uCPUID = 0U;                       //定义存放 CPU ID 的数组
05       asm                                    //开始执行汇编
06      {
07          mov eax, 0                          //获取 CPU 型号
08              cpuid
09              mov dword ptr szCPU[0], ebx     //获取 CPU 型号的前 4 个字符
10              mov dword ptr szCPU[4], edx     //获取 CPU 型号的中间 4 个字符
11              mov dword ptr szCPU[8], ecx     //获取 CPU 型号的最后 4 个字符
12              mov eax, 1                      //获取 CPU ID
13              cpuid
14              mov uCPUID, edx                 //获取 CPU ID 的值
15      }
16      //输出 CPU 信息
17      WriteLog("当前系统的 CPU 类型为=%s----CPU ID=%u", szCPU, uCPUID);
18  }
```

上面代码首先使用 mov 语句设置 eax 的值为 0，表示功能为获取 CPU 公司信息，调用 cpuid 指令获取 CPU 生产厂家，并将获取的值存入字符数组 szCPU 中。然后使用 mov 语句设置 eax 的值为 1，表示功能为获取 CPU ID，调用 cpuid 指令获取 CPU ID，并将获取的值存入 UINT 类型的变量 uCPUID 中。最后将这两个信息显示出来，运行效果如图 20-49 所示。

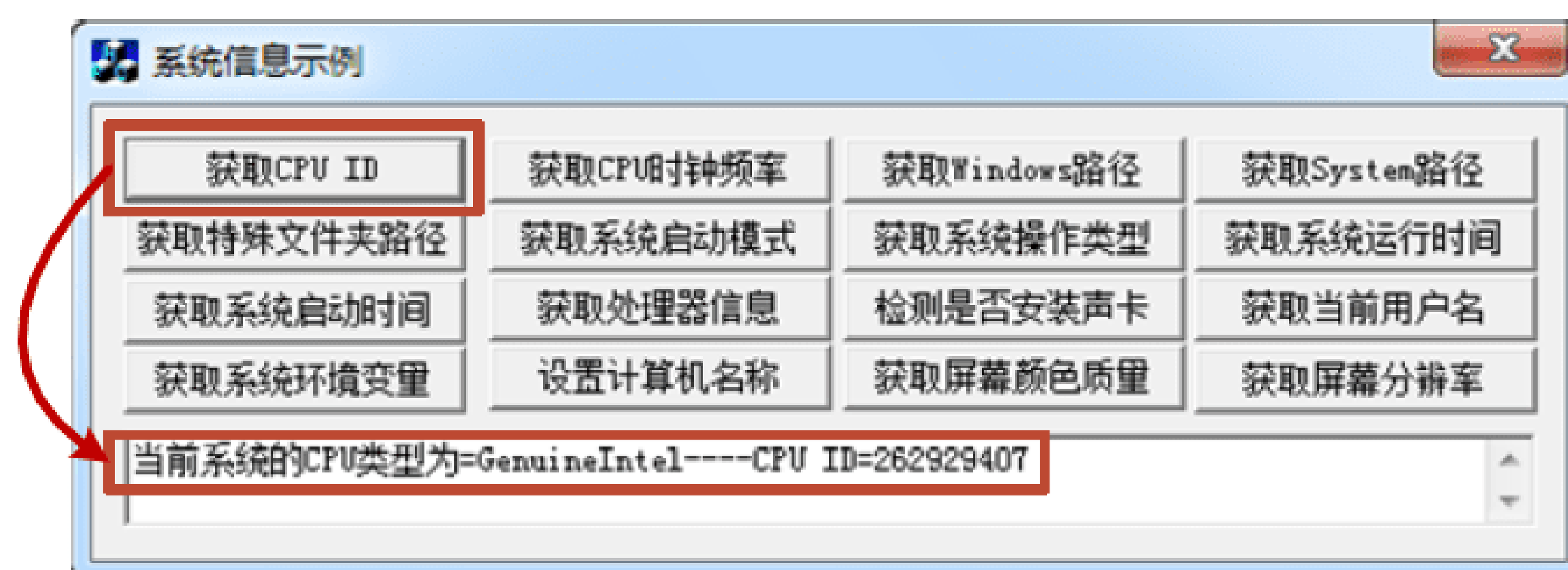


图 20-49　获取 CPU ID 运行效果

20.7.2　获取 CPU 时钟频率

除了可以使用汇编语句从系统中获取有关硬件的信息，还可以通过注册表获取硬件信息。如可以通过读取注册表数据获取 CPU 时钟频率。代码如下：

```
01  void CSystemInfoSampleDlg::OnButtonGetCpufrequency()//获取 CPU 时钟频率
02  {
03      unsigned long ulSpeed=0;                        //定义 CPU 时钟频率变量
04      HKEY hKey;                                      //定义注册表键
05      if RegOpenKeyEx(HKEY_LOCAL_MACHINE,
06          "HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0",
07          0,KEY_READ,&hKey)==ERROR_SUCCESS)            //打开注册表
08      {
09          unsigned long ulLen= sizeof(ulSpeed);       //赋值长度
10          //查询 CPU 时钟频率
11          RegQueryValueEx(hKey, "~MHz", NULL, NULL,
12                          (LPBYTE)&ulSpeed, &ulLen);
13          RegCloseKey(hKey);                          //关闭注册表
```

```
14          WriteLog("CPU 时钟频率＝%ldMHz",ulSpeed); //显示获取的信息
15      }
16      else
17          WriteLog("获取 CPU 时钟频率失败");            //输出错误信息
18  }
```

上面代码读取 HARDWARE\DESCRIPTION\System\CentralProcessor\0 注册表项的~MHz 值，其中就存储 CPU 时钟频率。有关注册表读写的操作会在第 22 章中详细介绍，程序运行的效果如图 20-50 所示。

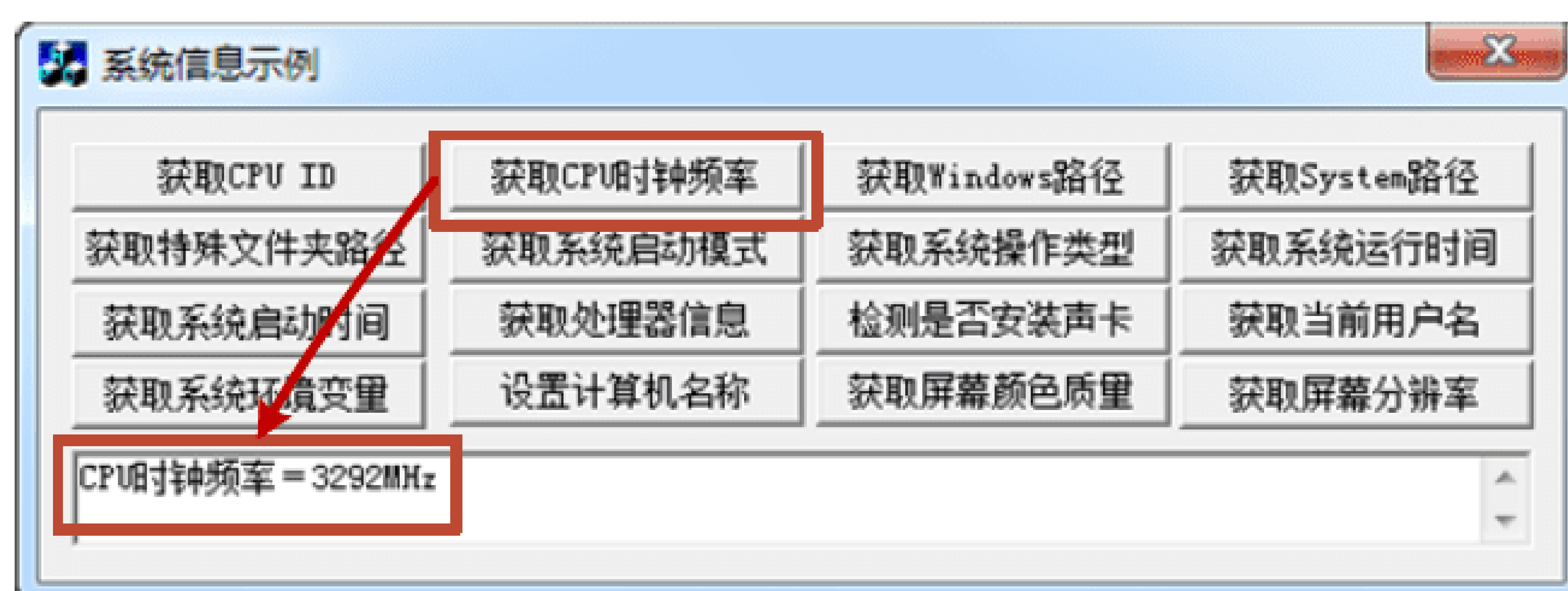


图 20-50　获取 CPU 时钟频率的运行效果

20.7.3　获得 Windows 和 System 的路径

Windows 中有两个比较重要的目录：Windows 目录和 System32 目录。Windows 目录中包含诸如 Win32 应用程序、初始化文件和帮助文件；System32 目录中包含诸如动态链接库、驱动和字体文件，系统不允许应用程序在此目录中创建文件。Windows 提供了 GetWindowsDirectory()函数和 GetSystemDirectory()函数分别可以获取这两个目录。其函数原型如下：

```
UINT GetWindowsDirectory(
  LPTSTR lpBuffer,  //指向包含路径的以 NULL 结束的字符串的缓冲区
  UINT uSize );     //指定缓冲区的最大字节数，应该最少设置为 MAX_PATH
UINT GetSystemDirectory(
  LPTSTR lpBuffer,  //指向包含路径的以 NULL 结束的字符串的指针
  UINT uSize);      //指定缓冲区的最大字节数，应该最少设置为 MAX_PATH
```

上面两个函数，如果执行成功，则返回值为缓冲区中的字节数，并且将获取的路径信息存放在 lpBuffer 变量中。以下代码所示为获取 Windows 路径和 System 路径的方法。

```
01  void CSystemInfoSampleDlg::OnButtonGetWindowpath() //获得 Windows 路径
02  {
03      TCHAR szPath[MAX_PATH]={0};                    //定义路径变量
04      //获取 Windows 路径
05      int nLength = GetWindowsDirectory(szPath, MAX_PATH);
06      //输出路径信息
07      if (nLength > 0)
08          WriteLog("获取 Window 路径=%s", szPath);
09      else
10          WriteLog("获取 Window 路径失败");           //输出错误信息
11  }
12  void CSystemInfoSampleDlg::OnButtonGetSystempath() //获得 System 路径
13  {
14      TCHAR szPath[MAX_PATH]={0};                    //定义路径变量
```

```
15      //获取 System 路径
16      int nLength =  GetSystemDirectory(szPath, MAX_PATH);
17      //输出路径信息
18      if (nLength > 0)
19          WriteLog("获取 System 路径=%s", szPath);
20      else
21          WriteLog("获取 System 路径失败");                 //输出错误信息
22  }
```

上面代码使用 GetWindowsDirectory 函数和 GetSystemDirectory()函数获取了 Windows 目录和系统目录，并将其存储在 szPath 变量中。程序运行效果如图 20-51 所示。

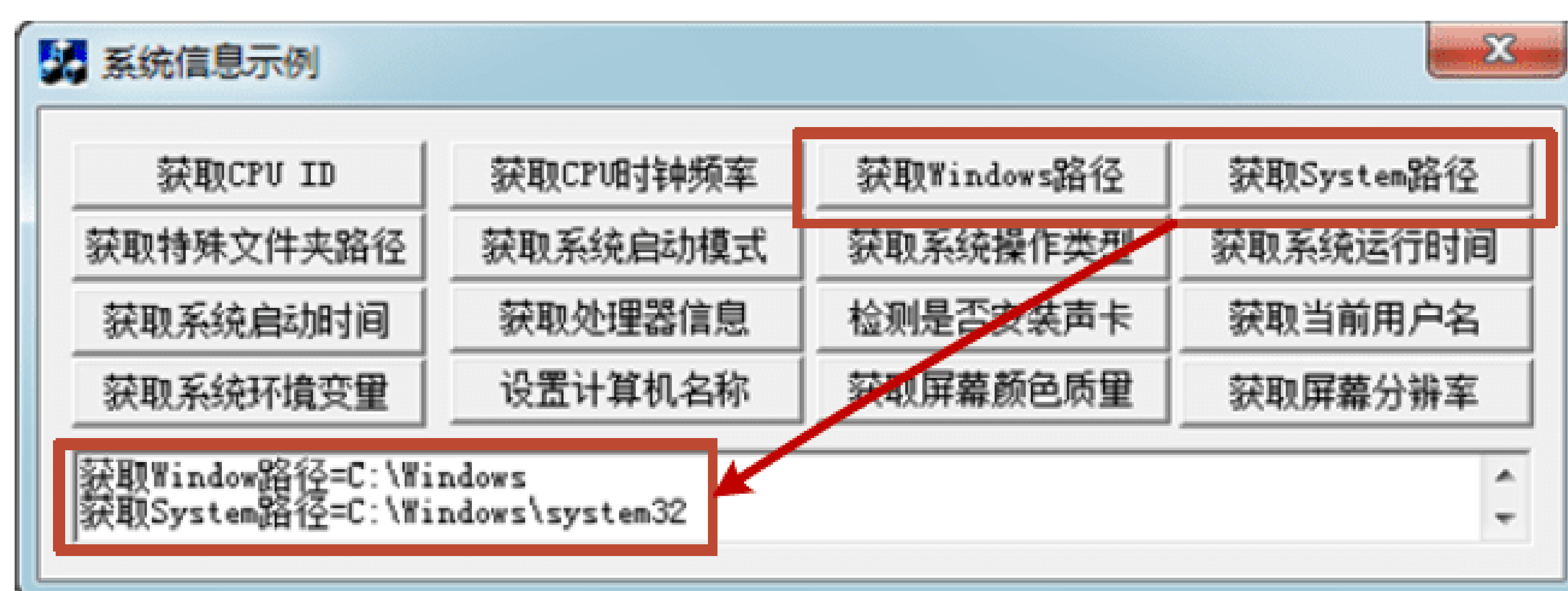


图 20-51　获取 Windows 和 System 路径运行效果

20.7.4　获取特殊文件夹路径

调用 Windows API 函数 SHGetSpecialFolderLocation()可以获取特殊文件夹的路径。其函数原型为：

```
WINSHELLAPI HRESULT WINAPI SHGetSpecialFolderPath(
  HWND hwndOwner,         //表示要将其显示的对话框的父窗体句柄
  LPTSTR lpszPath,        //存放获取的路径值的变量
  int nFolder,            //要获取的特殊文件夹的种类
  BOOL fCreate);          //表示如果要获取的路径不存在，是否自动创建
```

函数的 nFolder 参数表示要获取的特殊文件夹的种类，其有效取值如表 20-9 所示。

表 20-9　特殊文件夹的种类

Index 值	特殊文件夹的含义
CSIDL_ALTSTARTUP	用户的非本地化启动程序组路径
CSIDL_APPDATA	应用程序数据路径
CSIDL_BITBUCKET	回收站的路径
CSIDL_COMMON_ALTSTARTUP	所有用户的非本地化启动程序组路径
CSIDL_COMMON_DESKTOPDIRECTORY	所有用户的桌面路径
CSIDL_COMMON_FAVORITES	所有用户的收藏夹路径
CSIDL_COMMON_PROGRAMS	所有用户的启动菜单程序路径
CSIDL_COMMON_STARTMENU	所有用户的启动菜单路径
CSIDL_COMMON_STARTUP	所有用户的启动文件夹路径
CSIDL_CONTROLS	控制面板程序的路径
CSIDL_COOKIES	存放 cookies 的路径

续表

Index 值	特殊文件夹的含义
CSIDL_DESKTOP	Windows 桌面
CSIDL_DESKTOPDIRECTORY	存放桌面文件对象的目录
CSIDL_DRIVES	我的电脑文件夹
CSIDL_FAVORITES	用户收藏夹文件夹
CSIDL_FONTS	字体文件夹
CSIDL_HISTORY	Internet 历史文件夹
CSIDL_INTERNET	Internet 文件夹
CSIDL_INTERNET_CACHE	Internet 缓冲区文件夹
CSIDL_NETHOOD	网络邻居文件夹
CSIDL_NETWORK	网络邻居文件夹，表示网络顶层架构
CSIDL_PERSONAL	我的文档文件夹
CSIDL_PRINTERS	打印机文件夹
CSIDL_PRINTHOOD	网络打印机文件夹
CSIDL_PROGRAMS	用户应用程序组文件夹
CSIDL_RECENT	用户最近打开的文档文件夹
CSIDL_SENDTO	发送到菜单项的文件夹
CSIDL_STARTMENU	用户的启动菜单项文件夹
CSIDL_STARTUP	用户的启动文件夹
CSIDL_TEMPLATES	临时文档文件夹

下面的代码显示了如何获取特殊文件夹，读者可以根据自己的需要获取特殊文件夹。

```
01  //获取特殊文件夹路径
02  void CSystemInfoSampleDlg::OnButtonGetspecialpath()
03  {
04      TCHAR  szPath[MAX_PATH];                         //定义路径变量
05      //路径 CSIDL 数组
06      int  iIndex[]={CSIDL_ALTSTARTUP,CSIDL_APPDATA,CSIDL_BITBUCKET,
07          CSIDL_COMMON_ALTSTARTUP,CSIDL_COMMON_DESKTOPDIRECTORY,
08          CSIDL_COMMON_FAVORITES,CSIDL_COMMON_PROGRAMS,
09          CSIDL_COMMON_STARTMENU,CSIDL_COMMON_STARTUP,CSIDL_CONTROLS,
10          CSIDL_COOKIES,CSIDL_DESKTOP, CSIDL_DESKTOPDIRECTORY,
11          CSIDL_DRIVES,CSIDL_FAVORITES,CSIDL_FONTS,CSIDL_HISTORY,
12          CSIDL_INTERNET,CSIDL_INTERNET_CACHE,CSIDL_NETHOOD,
13          CSIDL_NETWORK, CSIDL_PERSONAL,CSIDL_PRINTERS,
14          CSIDL_PRINTHOOD,CSIDL_PROGRAMS, CSIDL_RECENT,CSIDL_SENDTO,
15          CSIDL_STARTMENU,CSIDL_STARTUP,CSIDL_TEMPLATES};
16      CString csIndex[]={"CSIDL_ALTSTARTUP", "CSIDL_APPDATA",
17          "CSIDL_BITBUCKET","CSIDL_COMMON_ALTSTARTUP",
18          "CSIDL_COMMON_DESKTOPDIRECTORY",
19          "CSIDL_COMMON_FAVORITES", "CSIDL_COMMON_PROGRAMS",
20          "CSIDL_COMMON_STARTMENU", "CSIDL_COMMON_STARTUP",
21          "CSIDL_CONTROLS", "CSIDL_COOKIES","CSIDL_DESKTOP",
22          "CSIDL_DESKTOPDIRECTORY", "CSIDL_DRIVES",
23          "CSIDL_FAVORITES","CSIDL_FONTS", "CSIDL_HISTORY",
24          "CSIDL_INTERNET","CSIDL_INTERNET_CACHE",
25          "CSIDL_NETHOOD", "CSIDL_NETWORK", "CSIDL_PERSONAL",
26          "CSIDL_PRINTERS", "CSIDL_PRINTHOOD","CSIDL_PROGRAMS",
27          "CSIDL_RECENT","CSIDL_SENDTO", "CSIDL_STARTMENU",
```

```
28          "CSIDL_STARTUP","CSIDL_TEMPLATES"}; //路径名称数组
29      for (int i = 0;i < 30; i++)          //使用 for 循环依次获取文件夹
30      {
31          memset(szPath, 0x00, sizeof(szPath));//清空变量值
32          //获取特殊文件夹
33          if (SHGetSpecialFolderPath(HWND_DESKTOP, szPath,
34                      iIndex[i],false) != 0)
35              WriteLog("%s=%s", csIndex[i], szPath);//输出文件夹路径
36          else
37              WriteLog("获取%s 失败", csIndex[i]);//输出错误信息
38      }
39  }
```

上面代码运行的效果如图 20-52 所示。



图 20-52　获取特殊文件夹路径运行效果

20.7.5　检测系统启动模式

Windows API 函数 GetSystemMetrics()可以获取多种有关系统的信息，其函数原型为：

```
int GetSystemMetrics(
  int nIndex  );          //要获取的系统信息的索引值，必须设置为 SM_CLEANBOOT
```

此函数的返回值即为获取的系统启动模式，以下代码是此函数的使用方法。

```
01  void CSystemInfoSampleDlg::OnButtonGetstartmode()  //检测系统启动模式
02  {
03      int iMode = GetSystemMetrics(SM_CLEANBOOT);     //获取系统启动模式
04      switch (iMode)                                  //判断返回的模式值
05      {
06      case 0:                                         //正常模式
07          WriteLog("系统启动模式为--正常模式");
08          break;
09      case 1:                                         //安全模式
10          WriteLog("系统启动模式为--安全模式");
11          break;
12      case 2:                                         //网络环境下安全模式
13          WriteLog("系统启动模式为--网络环境下安全模式");
14          break;
15      default:                                        //其他模式
```

```
16          WriteLog("系统启动模式为--其他");
17          break;
18      }
19  }
```

上面代码使用 GetSystemMetrics()函数传入 SM_CLEANBOOT 参数获取系统的启动模式，并通过判断整型返回值显示系统的启动模式。运行的效果如图 20-53 所示。

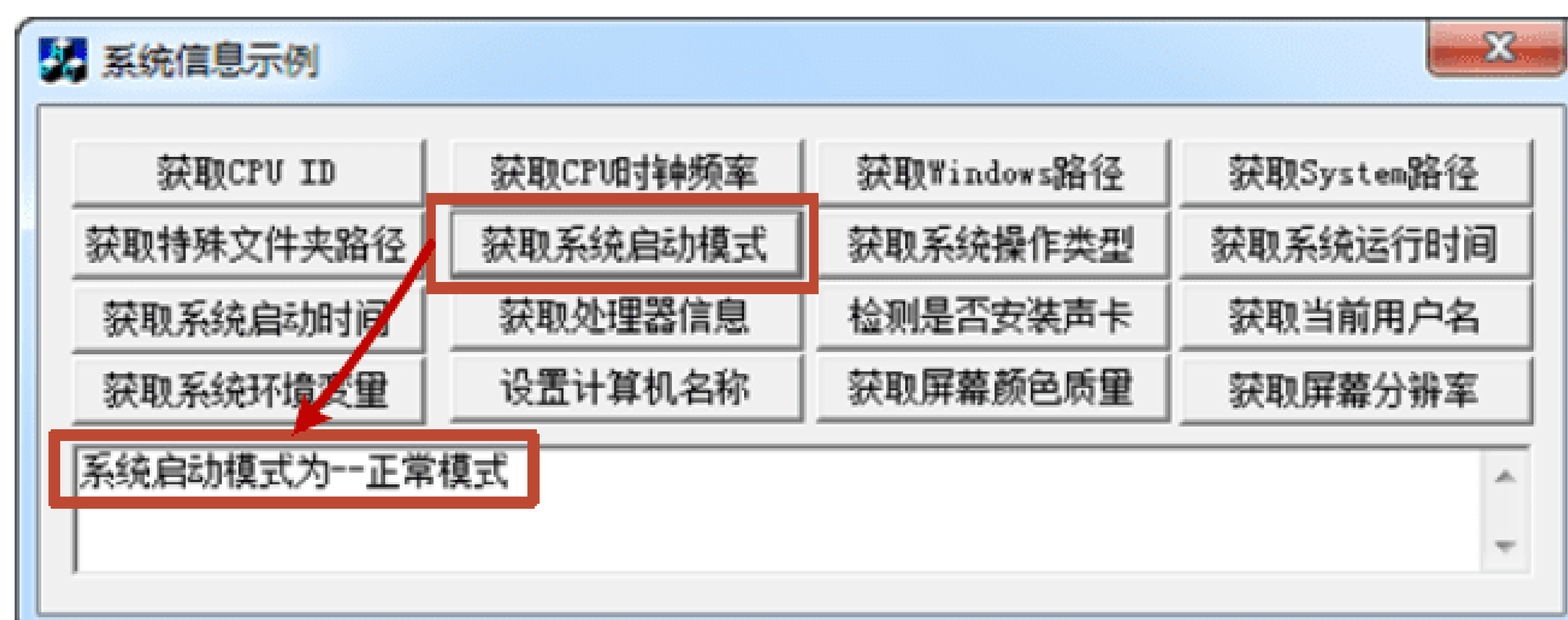


图 20-53　获取系统启动模式的运行效果

20.7.6　判断操作系统类型

通过 GetVersionEx()函数可以判断当前运行的操作系统的版本信息。其函数原型为：

```
BOOL GetVersionEx(
  LPOSVERSIONINFO lpVersionInformation );//OSVERSIONINFO 变量
```

其中，lpVersionInformation 参数是指向包含操作系统版本信息的 OSVERSIONIFO 数据结构变量的指针，包括主版本号和次版本号、生成版本号、平台标识符和操作系统的描述。结果定义如下：

```
typedef struct  OSVERSIONINFO{
  DWORD dwOSVersionInfoSize;       //指定此数据结构的字节数
  DWORD dwMajorVersion;            //指定操作系统的主版本号
  DWORD dwMinorVersion;            //指定操作系统的次版本号
  DWORD dwBuildNumber;             //指定操作系统的生成版本号
  DWORD dwPlatformId;              //指定操作系统的平台 ID
  TCHAR szCSDVersion[ 128 ];       //指定操作系统的描述
} OSVERSIONINFO;
```

下面代码调用 GetVersionEx()函数获取了当前操作系统的版本。

```
01  //判断操作系统类型
02  void CSystemInfoSampleDlg::OnButtonGetSysversion()
03  {
04      OSVERSIONINFO osvi;                                   //定义版本信息结构
05      //设置版本结构大小
06      osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
07      if (GetVersionEx (&osvi))                             //获取版本信息
08      //输出版本信息
09          WriteLog("获取操作系统版本成功，主版本=%d;次版本=%d;\r\n 生成版本=%d;
10          平台 ID=%d;系统描述=%s",osvi.dwMajorVersion,osvi.dwMinorVersion,
```

```
11          osvi.dwBuildNumber,osvi.dwPlatformId,osvi.szCSDVersion);
12      else
13          WriteLog("获取操作系统版本失败");                //输出错误信息
14  }
```

上面代码使用 GetVersionEx()函数获取了操作系统的版本信息，选取了其中的主版本号、次版本号、生成版本号和平台 ID 以及系统的描述，并在界面上显示出来。运行的效果如图 20-54 所示。

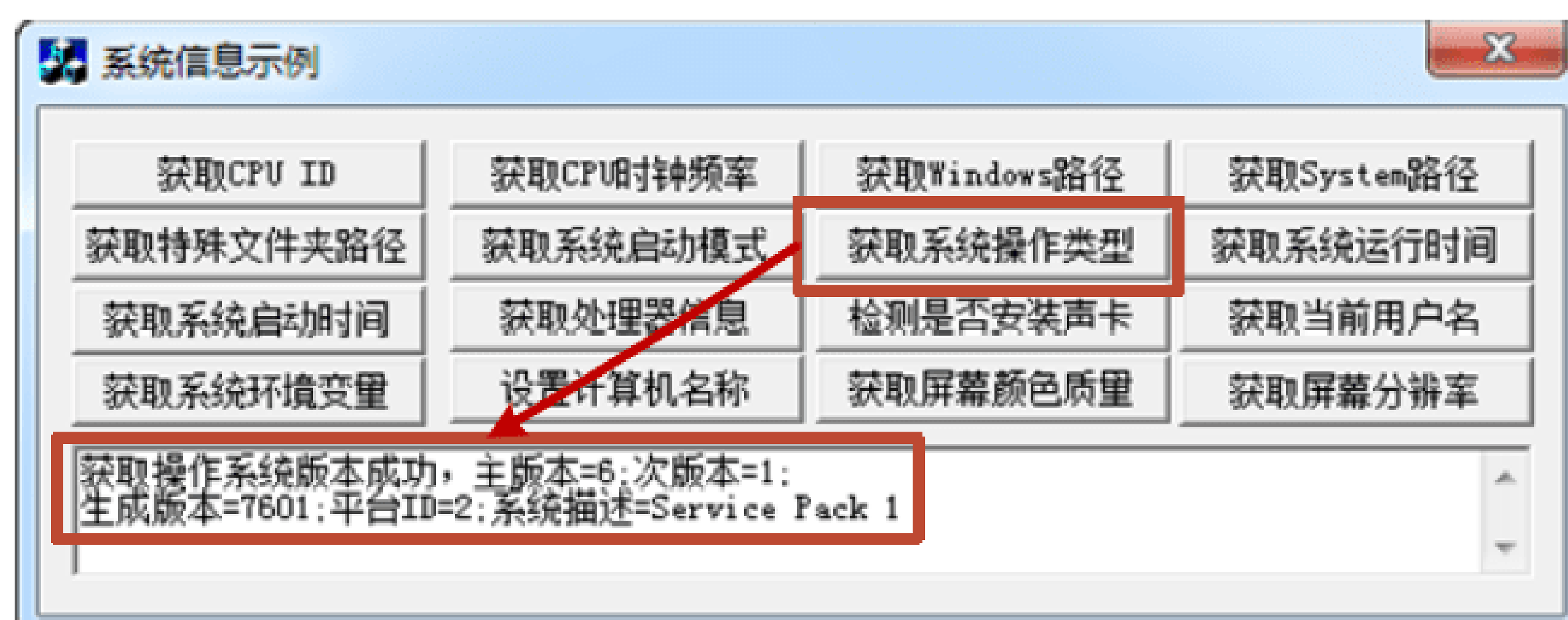


图 20-54　获取操作系统类型运行效果

20.7.7　获取当前系统的运行时间

调用 Windows API 函数 GetTickCount()可以获取当前系统运行时间的毫秒数，通过将毫秒数换算成时间值，可以确定当前系统的运行时间。代码如下：

```
01  //获取当前系统的运行时间
02  void CSystemInfoSampleDlg::OnButtonGetruntime()
03  {
04      DWORD dwTicks = GetTickCount();       //获取运行的单位时间数
05      CTimeSpan timeSpan(dwTicks/1000);     //将其转换为 CTimeSpan 类型
06      //输出运行时间信息
07      WriteLog("系统已经运行了%d 天，%d 小时，%d 分钟，%d 秒",
08          timeSpan.GetDays(),timeSpan.GetHours(),
09          timeSpan.GetMinutes(), timeSpan.GetSeconds());
10  }
```

上面代码通过 GetTickCount()函数获取当前系统的运行时间，然后将其赋值给 CTimeSpan 类型的变量，换算成时间间隔类，最后将其值格式化成字符串显示在界面上。程序运行的效果如图 20-55 所示。

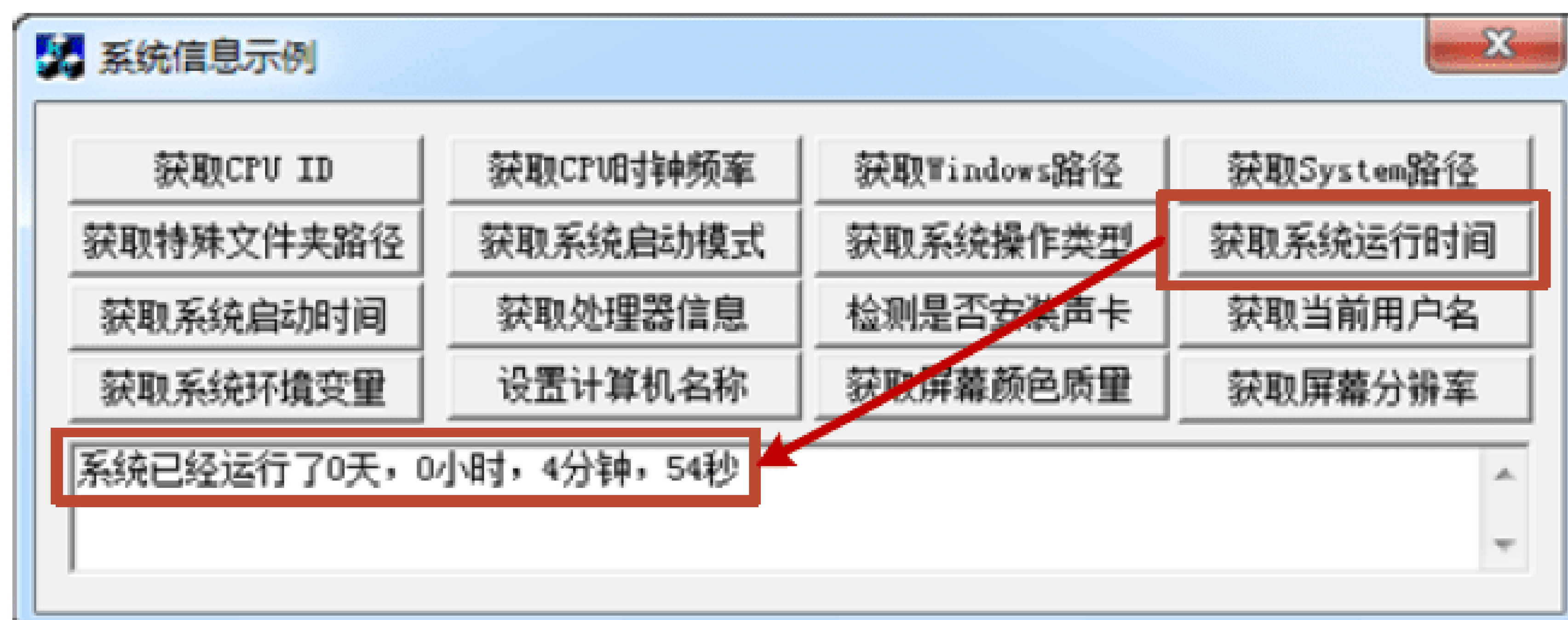


图 20-55　获取当前系统的运行时间效果

20.7.8　如何获取 Windows 7 系统启动时间

调用 Windows API 函数 GetTickCount()可以获取当前系统运行时间的毫秒数，通过将毫秒数与当前时间进行运算，可以确定系统的启动时间。代码如下：

```
01  void CSystemInfoSampleDlg::OnButtonGetstarttime()
02  {
03      DWORD dwTicks = GetTickCount();            //获取运行的单位时间数
04      CTime time = CTime::GetCurrentTime();      //获取当前时间
05      CTimeSpan timeSpan(dwTicks/1000);          //将其转换为 CTimeSpan 类型
06      time -= timeSpan;                          //当前时间减去已经运行的时间
07      //输出启动时间
08      WriteLog("系统启动时间为：%s", time.Format("%Y-%m-%d %H:%M:%S"));
09  }
```

上面代码首先调用 GetTickCount()函数获取系统已经运行时间的毫秒数，然后将其换算成时间间隔类 CTimeSpan 的变量，在当前时间值上减去运行的时间值，得到的结果就是系统的启动时间。程序运行的效果如图 20-56 所示。



图 20-56　获取当前系统的启动时间效果

20.7.9　获取处理器信息

在 Windows 平台下，通过 GetSystemInfo()函数可以获取有关系统的信息，主要是处理器的信息。此函数没有返回值，将结果信息存入 SYSTEM_INFO 结构的变量中。函数原型为：

```
VOID GetSystemInfo(
  LPSYSTEM_INFO lpSystemInfo );     //指向 SYSTEM_INFO 结构的变量的指针
```

以下代码为此函数的调用方法。

```
01  void CSystemInfoSampleDlg::OnButtonGetProcessor()  //获取处理器信息
02  {
03      SYSTEM_INFO sysinfo;                           //定义系统信息结构
04      GetSystemInfo(&sysinfo);                       //获取处理器信息
05      switch(sysinfo.wProcessorArchitecture)         //判断处理器种类
06      {
07      case PROCESSOR_ARCHITECTURE_INTEL:             //Intel 处理器
08          switch (sysinfo.wProcessorLevel)           //判断处理器型号
09          {
```

```
10          case 3:
11              WriteLog("处理器类型=
12                          PROCESSOR_ARCHITECTURE_INTEL--Intel 80386");
13              break;
14          case 4:
15              WriteLog("处理器类型=
16                          PROCESSOR_ARCHITECTURE_INTEL--Intel 80486");
17              break;
18          case 5:
19              WriteLog("处理器类型=
20                               PROCESSOR_ARCHITECTURE_INTEL--Pentium");
21              break;
22          default:
23              WriteLog("处理器类型=PROCESSOR_ARCHITECTURE_INTEL");
24              break;
25          }
26          break;
27      case PROCESSOR_ARCHITECTURE_MIPS :                //MIPS
28          WriteLog("处理器类型=MIPS--R%d000", sysinfo.wProcessorLevel);
29          break;
30      case PROCESSOR_ARCHITECTURE_ALPHA:                //ALPHA
31          WriteLog("处理器类型=ALPHA--%d", sysinfo.wProcessorLevel);
32          break;
33      case PROCESSOR_ARCHITECTURE_PPC:                  //PPC
34          switch (sysinfo.wProcessorLevel)              //判断处理器型号
35          {
36          case 1:
37              WriteLog("处理器类型=PPC--PPC 601");
38              break;
39          case 3:
40              WriteLog("处理器类型=PPC--PPC 603");
41              break;
42          case 4:
43              WriteLog("处理器类型=PPC--PPC 604");
44              break;
45          case 6:
46              WriteLog("处理器类型=PPC--PPC 603+");
47              break;
48          case 9:
49              WriteLog("处理器类型=PPC--PPC 604+");
50              break;
51          case 20:
52              WriteLog("处理器类型=PPC--PPC 620");
53              break;
54          default:
55              WriteLog("处理器类型=PPC--PPC PPC");
56          }
57          break;
58      case PROCESSOR_ARCHITECTURE_UNKNOWN :             //处理器未知
59          WriteLog("处理器类型未知");
60          break;
61      default:                                          //其他未知值
62          WriteLog("处理器类型未知");
63          break;
64      }
65      //输出处理器数目
66      WriteLog("系统中的处理器数目为%d",sysinfo.dwNumberOfProcessors);
67  }
```

上面代码从 SYSTEM_INFO 结构中获取处理器的类型和处理器的数目，有关其他信息，读者可以自行读取。程序运行的效果如图 20-57 所示。

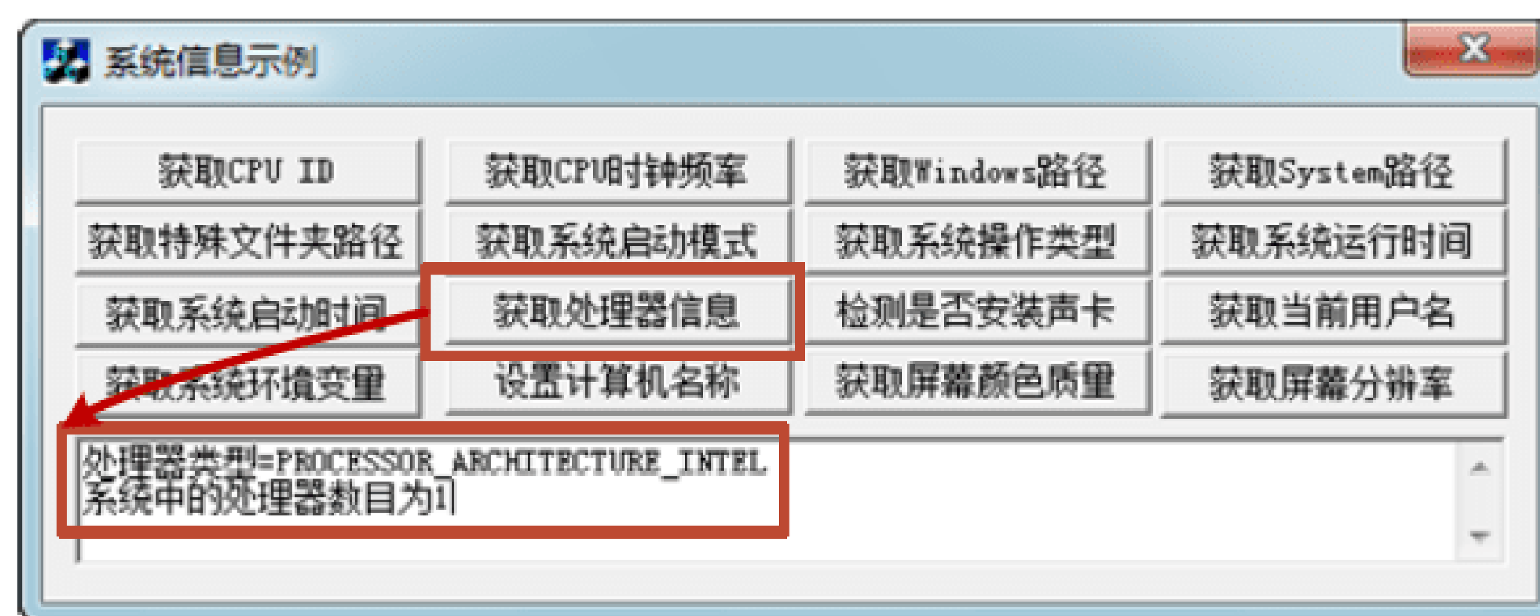


图 20-57　获取处理器信息运行效果

20.7.10　检测是否安装声卡

在 Windows 平台下，可以使用 waveOutOpen()函数检测系统中是否安装了声卡。其原来的作用是播放音频，可以通过传入指定的参数和判断返回值检测系统中是否安装了声卡。函数原型为：

```
MMRESULT waveOutOpen(
  LPHWAVEOUT phwo,                          //返回的播放句柄
  UINT uDeviceID,                           //指定设备 ID
  LPWAVEFORMATEX pwfx,                      //指定播放参数
  DWORD dwCallback,                         //指定播放后的回调函数
  DWORD dwCallbackInstance,                 //指定传入回调函数的参数
  DWORD fdwOpen);    //回调类型，WAVE_FORMAT_QUERY 检测系统中是否安装了声卡
```

下面代码检测是否安装了声卡。

```
01  void CSystemInfoSampleDlg::OnButtonIsaudio()//检测是否安装声卡
02  {
03      MMRESULT mmResult = waveOutOpen(NULL, WAVE_MAPPER, NULL,
04          NULL, NULL,WAVE_FORMAT_QUERY);        //打开音频播放
05      //如果错误结果为没有驱动相当于没有安装声卡
06      if (GetLastError() == MMSYSERR_NODRIVER)
07          WriteLog("系统中没有安装声卡");         //输出提示信息
08      else
09          WriteLog("系统中安装了声卡");           //输出提示信息
10  }
```

上面代码在调用了 waveOutOpen()函数后，判断 GetLastError()返回的最近一次错误代码是否为 MMSYSERR_NODRIVER。如果是，表示系统中没有安装声卡，如果不是，则表示系统中安装了声卡。程序运行效果如图 20-58 所示。

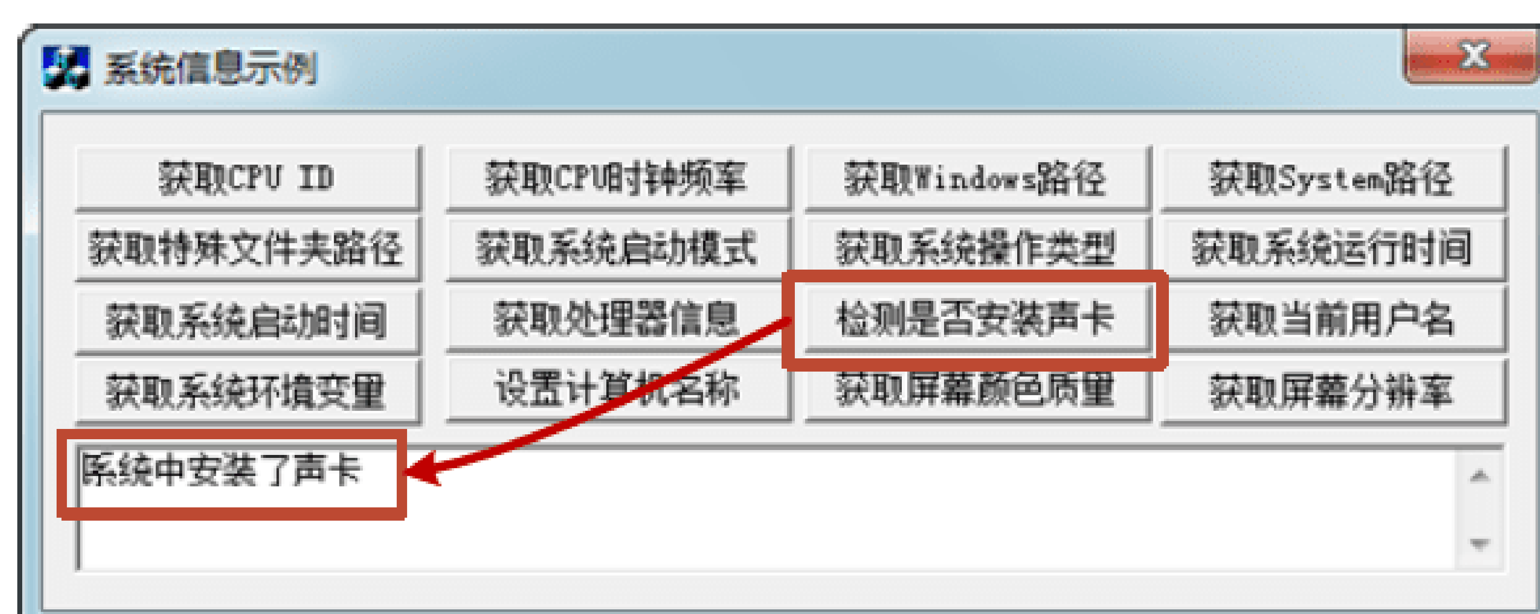


图 20-58　检测系统中是否安装声卡的运行效果

20.7.11　获取当前用户名

GetUserName()函数用于获取当前线程的用户名，即当前登录到系统中的用户名。

```
BOOL GetUserName(
  LPTSTR lpBuffer,             //指向接收用户登录名的缓冲区的指针
  LPDWORD nSize );             //表示缓冲区的最大长度
```

如果函数成功，则返回值为非 0，lpBuffer 中存放当前用户的登录名称。具体代码如下：

```
01  void CSystemInfoSampleDlg::OnButtonGetUsername()   //获取当前用户名
02  {
03      TCHAR szName[MAX_PATH];                         //定义用户名数组
04      DWORD dwLen = MAX_PATH;                         //定义用户名长度
05      if (GetUserName(szName, &dwLen))                //获取用户名
06          WriteLog("获取用户名成功,长度=%d;用户名=%s", dwLen, szName);
07      //输出用户名
08      else
09          WriteLog("获取用户名失败");                  //输出错误信息
10  }
```

上面代码中，使用 GetUserName()函数获取了当前的用户名，并将其存储在 szName 变量中，并在日志文本框中输出。程序运行效果如图 20-59 所示。

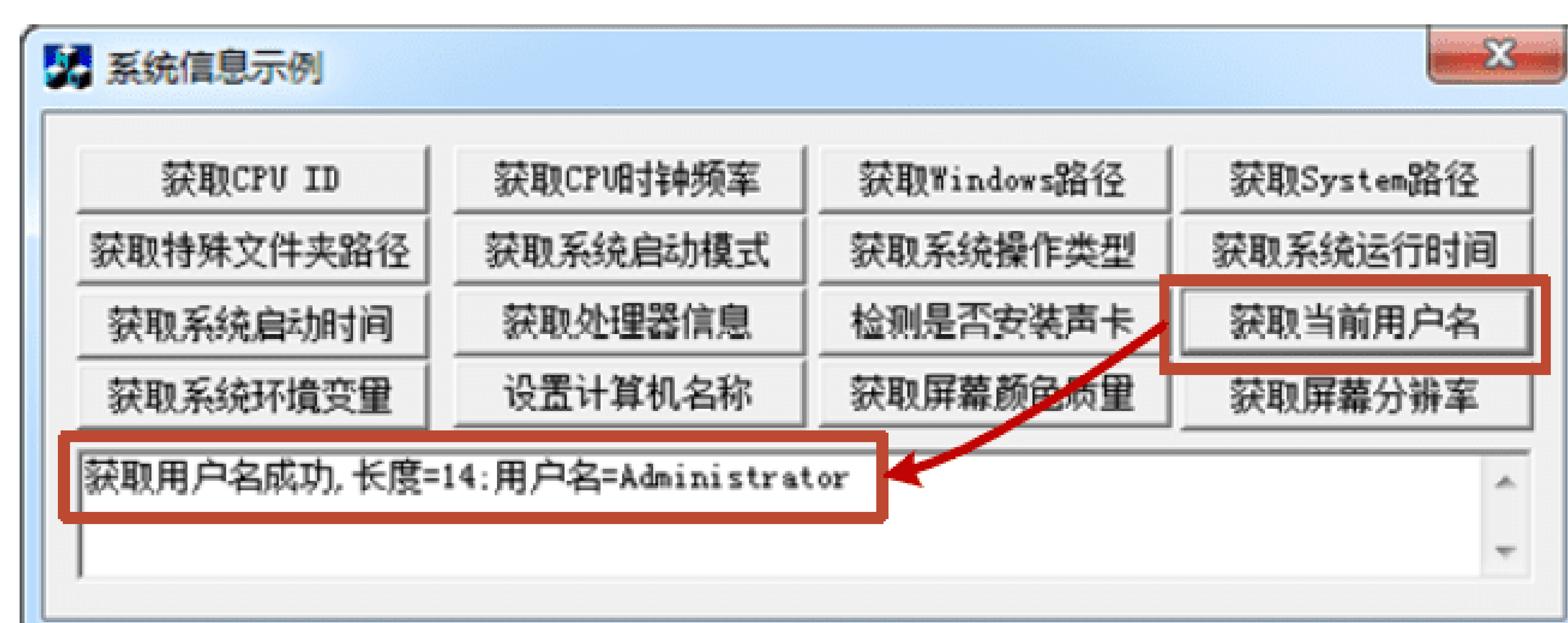


图 20-59　获取当前用户名的运行效果

20.7.12　获取系统环境变量

使用 GetEnvironmentVariable()函数，可以获取指定名称的环境变量，其函数原型为：

```
DWORD GetEnvironmentVariable(
  LPCTSTR lpName,                  //指定要获取的环境变量的名称
  LPTSTR lpBuffer,                 //存储获取的变量的值
  DWORD nSize );                   //指定获取的变量的值的长度
```

使用这个函数要获取系统环境变量，则需要指定 lpName 参数为 PATH，代码如下：

```
01  void CSystemInfoSampleDlg::OnButtonGetEnrovar()//获取系统环境变量
02  {
03      char szBuffer[1024] = {0};                  //定义环境变量值数组
04      DWORD dwSize=1024;                          //定义数组大小
05      //获取环境变量值
```

```
06      if (GetEnvironmentVariable("PATH",(LPTSTR)szBuffer, dwSize) > 0)
07          WriteLog("环境变量 PATH=%s",szBuffer);      //输出环境变量值
08      else
09          WriteLog("获取环境变量失败");               //输出错误提示
10  }
```

上面代码传入 PATH 指定要获取的系统环境变量，返回值表示获取的变量值的长度。如果该返回值大于 0，则显示出系统环境变量的值。程序运行效果如图 20-60 所示。

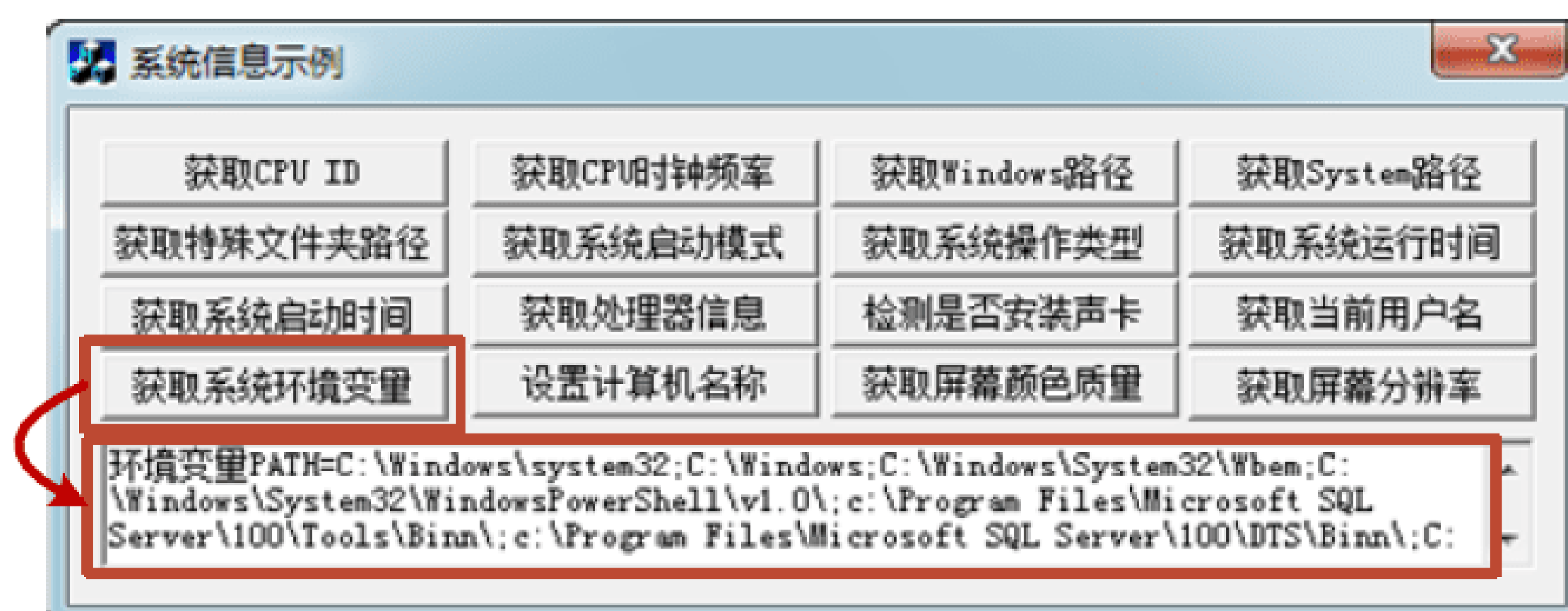


图 20-60　获取系统环境变量的运行效果

20.7.13　修改计算机名称

通过 SetComputerName()函数可以修改计算机名称。但是要使设置生效，需要重新启动计算机。同时，应用程序要使用此函数，必须具有 Administrator 的权限。其函数原型为：

```
BOOL SetComputerName(
  LPCTSTR lpComputerName);            //要设置的计算机名称的缓冲区指针
```

其中，lpComputerName 参数长度不能大于系统中规定的计算机名称最大值的定义 MAX_COMPUTERNAME_LENGTH。下面的代码是修改计算机名称的示例。

```
01  void CSystemInfoSampleDlg::OnButtonSetComputeName()//修改计算机名称
02  {
03      if (SetComputerName("ASUS-428EAA7DAB"))     //修改计算机名称
04      {
05          char szName[128];    //定义计算机名称数组
06          DWORD dwLen = 128;   //定义计算机名称长度
07          if (GetComputerName(szName, &dwLen))     //获取计算机名称
08              //输出计算机名称
09              WriteLog("设置计算机名称成功，修改后的计算机名称=%s",szName);
10      }
11      else
12          WriteLog("设置计算机名称失败");               //输出错误提示
13  }
```

在上面代码中，SetComputerName()函数的参数是要设置的计算机名称。如果设置成功，则调用 GetComputerName()函数获取当前计算机的名称。但是需要注意的是，计算机名称是在系统启动时从注册表中读取的，所以此时显示的计算机名称还是原来的名称。程序运行效果如图 20-61 所示。

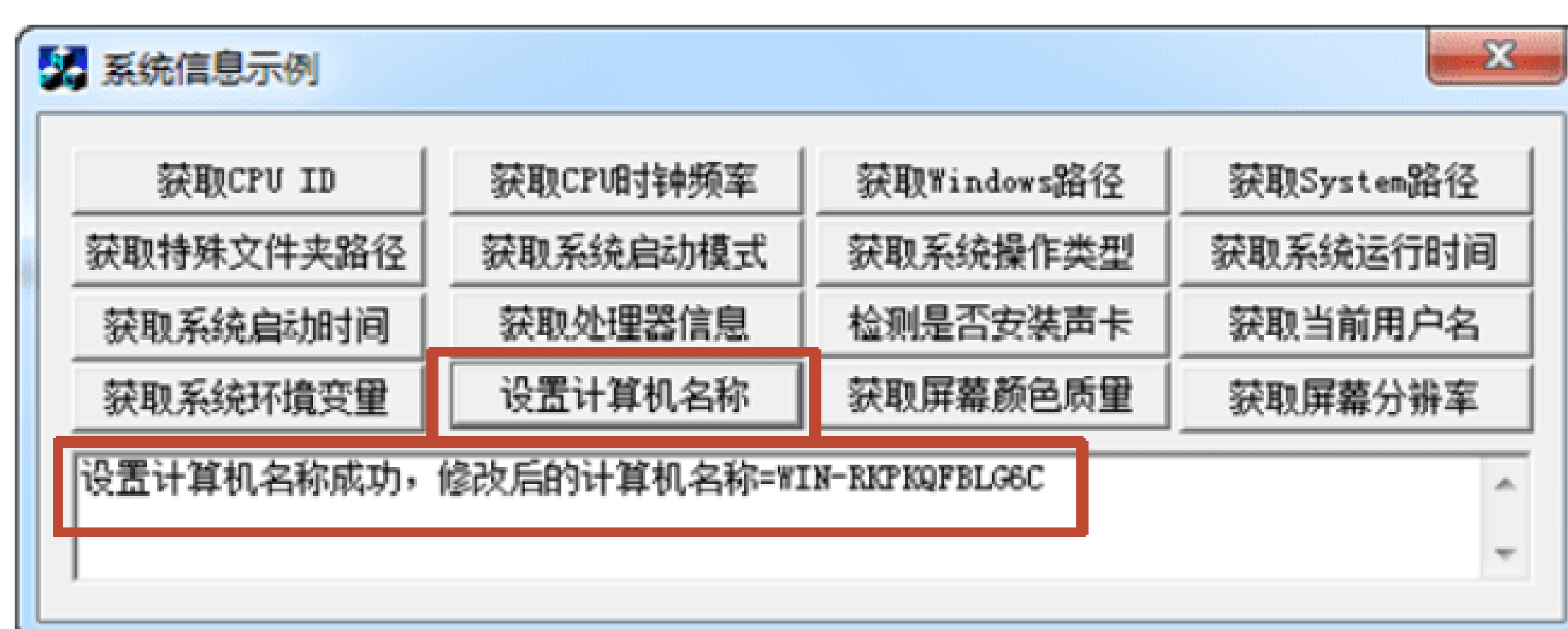


图 20-61　修改计算机名称运行效果

20.7.14　获取当前屏幕颜色质量

调用 GetDeviceCaps()函数可以获取指定设备上下文的相关信息。因此，可以使用此函数获取当前屏幕颜色的质量。其函数原型为：

```
int GetDeviceCaps(
  HDC hdc,                        //指定要获取信息的设备的句柄
  int nIndex );                   //要获取的信息种类
```

上面的函数使用 BITSPIXEL 获取屏幕颜色质量。代码如下：

```
01  void CSystemInfoSampleDlg::OnButtonGetScreencolor()
02  {
03      HDC hdc = GetDC()->m_hDC;                      //获取设备上下文句柄
04      int iColors = GetDeviceCaps(hdc, BITSPIXEL);//获取屏幕颜色质量
05      WriteLog("当前屏幕颜色质量为%d", iColors);      //输出屏幕颜色质量
06  }
```

上面代码首先使用 GetDC()函数获取设备上下文，并将其传入 GetDeviceCaps()函数中。在此函数的第二个参数中指定 BITSPIXEL，表示要获取颜色质量，并将返回值显示出来。程序运行效果如图 20-62 所示。



图 20-62　获取屏幕颜色质量运行效果

20.7.15　获得当前屏幕的分辨率

前面介绍过使用 Windows API 函数 GetSystemMetrics()可以获取多种系统信息。如果传入 SM_CXSCREEN 参数和 SM_CYSCREEN 参数，则函数会分别返回显示器的 X 分辨率和 Y 分辨率，即宽方向的分辨率和高方向的分辨率。代码如下：

```
01  void CSystemInfoSampleDlg::OnButtonGetScreenxy()//获得当前屏幕的分辨率
02  {
03      int iScreenX = GetSystemMetrics(SM_CXSCREEN);//X 分辨率
04      int iScreenY = GetSystemMetrics(SM_CYSCREEN);//Y 分辨率
05      WriteLog("当前屏幕分辨率为%d*%d", iScreenX, iScreenY);
06                                                   //输出屏幕分辨率
07  }
```

上面代码分别使用参数 SM_CXSCREEN 和参数 SM_CYSCREEN 调用 GetSystem-Metrics()函数，并将返回的值显示出来。程序运行效果如图 20-63 所示。



图 20-63　获取当前屏幕的分辨率运行效果

20.8　消　　息

消息是 Windows 操作系统中实现数据处理流程转换的重要手段，操作系统中的许多操作都是通过将消息发送到消息队列中，按照顺序进行处理的方式实现的。因此，适当地在程序中使用消息，可以灵活地完成数据处理流程的转换。本节主要介绍如何自定义消息，如何向 Windows 注册消息、发送消息的两个函数之间的区别以及如何利用 WM_COPYDATA 函数实现进程间的数据传递。

20.8.1　如何自定义消息

在程序中，用户可以根据需要自定义消息完成数据流程之间的转换。在进程中自定义消息的格式如下：

```
#define WM_MY_MESSAGE    //消息整型值
```

虽然，可以任意指定消息的值，但是建议在自定义消息时，使用如下形式：

```
#define WM_MY_MESSAGE   WM_USER + 66
```

使用此种方式自定义消息可以避免与其他消息之间的冲突，因为 WM_USER 是系统为用户预留的消息标识值的起始值，用户可以在其基础上利用相对值定义新消息值。以下代码是自定义消息的使用。

```
01  #define WM_MY_MESSAGE   WM_USER + 66         //自定义消息
02  void CMessageSendSampleDlg::OnButtonSendMymessage()
03  {
```

```
04      SendMessage(WM MY MESSAGE, NULL, NULL);//发送自定义消息
05  }
06  LRESULT CMessageSendSampleDlg::WindowProc(UINT message,
07      WPARAM wParam, LPARAM lParam)              //程序处理函数
08  {
09      switch (message)                           //判断消息类型
10      {
11      case WM MY MESSAGE:
12          WriteLog("接收到 WM_MY_MESSAGE 消息");//接收到用户自定义消息，输出
13          break;
14      default:
15          break;
16      }
17      return CDialog::WindowProc(message, wParam, lParam);
18  }
```

在上面代码中，第一行自定义了 WM_MY_MESSAGE 消息，其值为 WM_USER+66。OnButtonSendMymessage()函数发送一条自定义 WM_MY_MESSAGE 消息。WindowProc()函数中判断消息类型，如果是自定义消息 WM_MY_MESSAGE，则在文本日志框中显示提示消息。程序运行效果如图 20-64 所示。

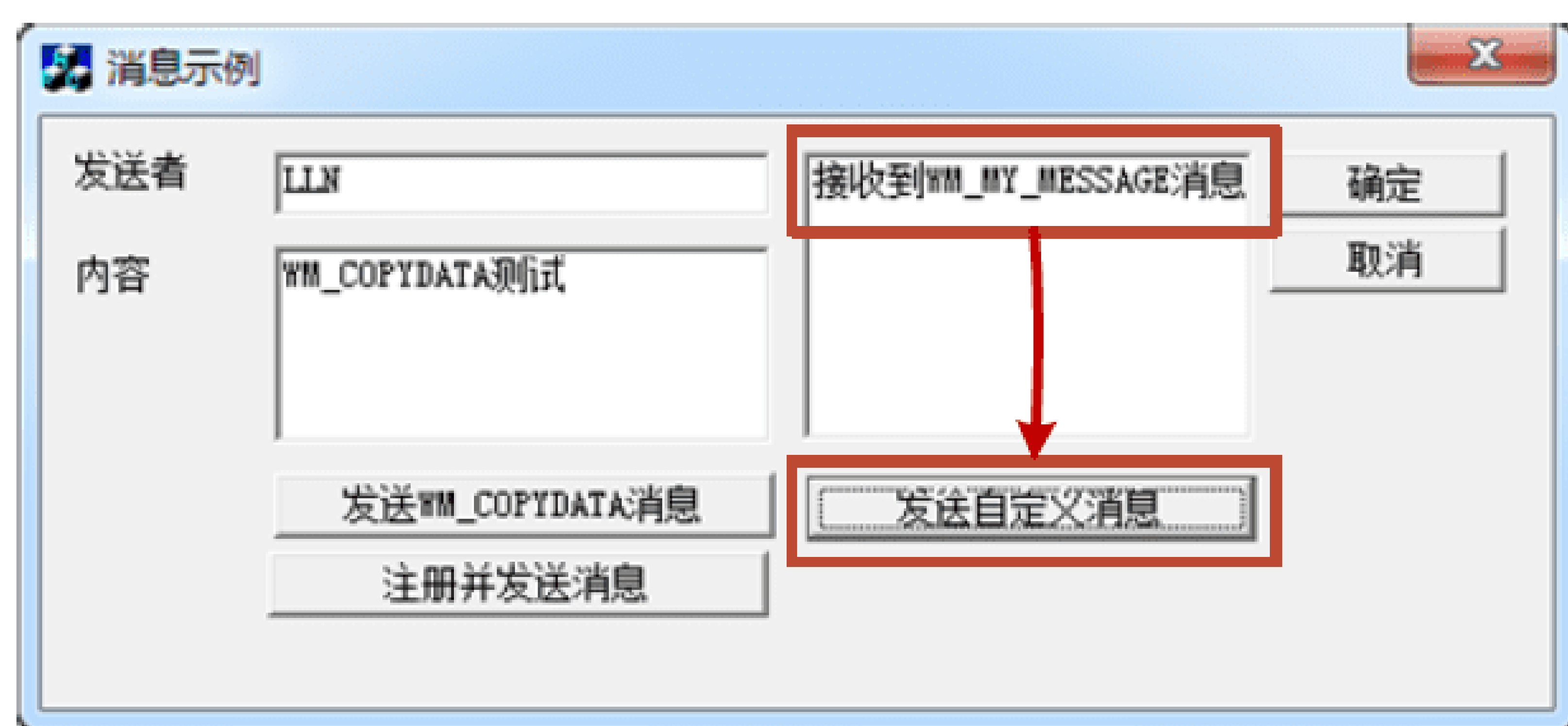


图 20-64　自定义消息运行效果

20.8.2　如何向 Windows 注册消息

RegisterWindowMessage()函数用于在 Windows 操作系统中注册指定名称的消息，使用此函数的返回值传入 SendMessage()函数和 PostMessage()函数可以发送消息。其函数原型为：

```
UINT RegisterWindowMessage(
  LPCTSTR lpString );                  //要注册的消息的字符串
```

此函数如果执行成功，会注册消息，返回值是 0xC000～0xFFFF 范围内的值。如果注册失败，则返回 0。通常此函数用于实现两个相关应用程序之间的通信，如果两个不同的应用程序注册相同的消息字符串，则返回给应用程序的消息值是相同的。通过这种方式，程序间可以通过相同名称的消息进行数据传递。代码如下：

```
01  void CMessageSendSampleDlg::OnButtonRegmessage()//注册消息函数
02  {
03      //注册消息
04      UINT myMsg = RegisterWindowMessage("LLN's Message Test");
```

```
05      //发送广播消息
06      ::SendMessage(HWND_BROADCAST, myMsg, NULL, NULL);
07  }
```

OnButtonRegmessage() 函数是发送消息的进程中的处理代码。首先调用 RegisterWindowMessage()函数向 Windows 注册指定字符串的消息，然后使用 SendMessage()函数广播此消息。下面是接收进程的处理代码：

```
01  BOOL CMessageReceiveSampleDlg::OnInitDialog()//对话框初始化函数
02  {
03  ...
04      myMsg = RegisterWindowMessage("LLN's Message Test");
05                                          //注册与发送进程相同的消息
06  ...
07  }
08  LRESULT CMessageReceiveSampleDlg::WindowProc(UINT message,
09       WPARAM wParam, LPARAM lParam)          //消息处理函数
10  {
11      if (message == myMsg)                   //如果接收到注册的消息
12          WriteLog("接收到使用 RegisterWindowMessage()函数注册的消息");
13                                          //输出提示信息
14      return CDialog::WindowProc(message, wParam, lParam);
15  }
```

上面是接收注册消息的进程的代码。在进程初始化函数中，首先要注册与发送进程注册的消息同名的消息，本例是在 OnInitDialog()函数中注册的，此处返回的 myMsg 值与 OnButton- Regmessage()函数中返回的 myMsg 的值是相同的。然后在接收进程的 WindowProc()函数中判断消息是否与注册消息返回的消息值相等，如果相等，表示接收到注册消息。当用户单击“注册并发送消息”按钮时，在接收程序的日志对话框中会显示出来。程序运行效果如图 20-65 所示。

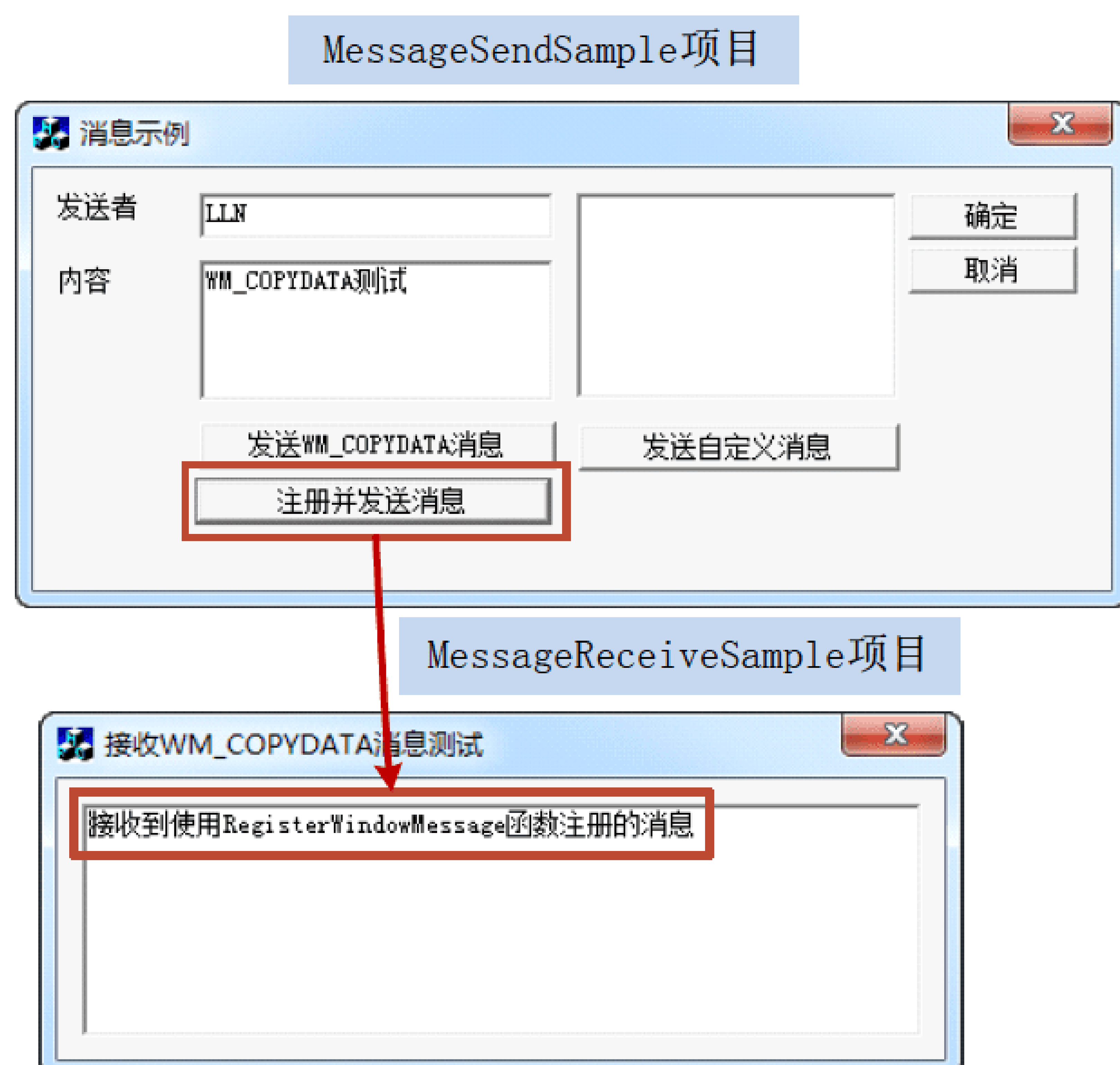


图 20-65　注册消息的使用运行效果

20.8.3　PostMessage()函数和 SendMessage()函数的区别

PostMessage()函数发送消息到与指定对话框相关的线程的消息队列中，并且不管线程是否处理消息，此函数都会立即返回。通过 GetMessage()函数和 PeekMessage()函数可以获取消息。函数原型为：

```
BOOL PostMessage(
  HWND hWnd,                    //指定接收消息的对话框的句柄
  UINT Msg,                     //指定发送的消息类型
  WPARAM wParam,                //消息参数
  LPARAM lParam );              //消息参数
```

如果 hWnd 参数指定为 HWND_BROADCAST，则消息会发送给所有的顶层对话框。

SendMessage()函数发送消息到指定的对话框，此函数直到目的对话框处理消息或发生错误时才会返回。其函数原型为：

```
LRESULT SendMessage(
  HWND hWnd,                        //指定接收消息的对话框的句柄
  UINT Msg,                         //指定发送的消息类型
  WPARAM wParam,                    //消息参数
  LPARAM lParam );                  //消息参数
```

如果 hWnd 参数指定为 HWND_BROADCAST，则消息会发送给所有的顶层对话框。此函数的返回值根据发送的消息类型和消息处理的结果而不同。

从上面的说明中可以看出，PostMessage()函数和 SendMessage()函数的主要区别在于，PostMessage()函数将消息发送到消息队列后，立即返回，不关心消息的处理情况；而 SendMessage()函数将消息发送给指定对话框后，会一直等待消息的处理结果。读者应该根据需要选择合适的消息函数发送消息。

20.8.4　利用 WM_COPYDATA 消息实现进程间数据传递

操作系统中提供预定义的 WM_COPYDATA 消息，将数据从一个进程发送到另一个进程中，实现进程间的数据传递。此消息传递的两个参数为：

```
wParam = (WPARAM) (HWND) hwnd;                  //发送对话框句柄
lParam = (LPARAM) (PCOPYDATASTRUCT) pcds;       //结构数据指针
```

其中 WPARAM 参数表示发送消息的对话框的句柄，LPARAM 参数是指向结构数据的指针，指向的数据结构为 PCOPYDATASTRUCT，其定义为：

```
typedef struct tagCOPYDATASTRUCT {
  DWORD dwData;                     //指定发送给接收程序的 32 位数据
  DWORD cbData;                     //指定 lpData 参数中指定的数据的长度
  PVOID lpData; } COPYDATASTRUCT;   //指向要传递的数据的指针
```

下面代码显示了如何通过 WM_COPYDATA 消息实现进程间的通信。

```
01  struct MESCONTENT                               //消息内容结构
02  {
03      char sender[50];                            //消息发送者
04      char content[500];                          //消息内容
05  };
06  //通过 WM_COPYDATA 消息发送数据
07  void CMessageSendSampleDlg::OnButtonSendCopydata()
08  {
09      UpdateData(true);                           //获取发送数据
10      //获取接收对话框
11      CWnd *pWnd=CWnd::FindWindow(NULL, _T("接收 WM_COPYDATA 消息测试"));
12      if(pWnd==NULL)                              //如果没有检索到接收对话框
13      {
14          WriteLog("接收消息数据的进程未运行");   //输出错误信息
15          return;                                 //返回
16      }
17      MESCONTENT msData={0};                      //定义消息结构变量
18      strcpy(msData.sender, m_sender.GetBuffer(50)); //赋值消息发送者
19      strcpy(msData.content,m_content.GetBuffer(500));//赋值消息内容
20      COPYDATASTRUCT cpd;                         //定义发送结构变量
21      cpd.dwData = 0;                             //发送整数值为 0
22      cpd.cbData = sizeof(msData);                //发送数据的长度
23      cpd.lpData = &msData;                       //发送数据内容
24      //发送 WM_COPYDATA 消息
25      pWnd->SendMessage(WM_COPYDATA,NULL, (LPARAM)&cpd);
26  }
```

上面代码首先定义了 MESCONTENT 结构，用于定义进程间传送的数据结构，此结构在发送进程和接收进程中的定义必须一致。OnButtonSendCopydata()函数是发送进程发送消息的代码。其中调用 FindWindow()函数查找接收此消息的进程，此处，使用第二个参数指定接收进程的窗体的标题来查找符合要求的进程对话框句柄，如果查找到相应句柄后，调用 SendMessage()函数发送 WM_COPYDATA 消息，将数据封装在 COPYDATASTRUCT 结构中。接收进程的接收代码如下：

```
01  BOOL CMessageReceiveSampleDlg::OnCopyData(CWnd* pWnd,
02      COPYDATASTRUCT* pCopyDataStruct)     //WM_COPYDATA 消息处理函数
03  {
04      //获取消息数据
05      MESCONTENT* msData = (MESCONTENT*)pCopyDataStruct->lpData;
06      WriteLog("接收到 WM_COPYDATA 消息。\r\n 发送者=%s;\r\n 内容=%s",
07          msData->sender, msData->content);//输出消息数据
08      return CDialog::OnCopyData(pWnd, pCopyDataStruct);
09  }
```

上面代码是接收进程处理 WM_COPYDATA 消息的处理函数，函数中将接收到的 COPYDATASTRUCT 类型的值进行解析处理，得到接收的内容，并在界面中显示出来。当在发送进程中单击“发送 WM_COPYDATA 消息”按钮时，在接收进程的日志文本框中会显示接收到的消息。程序运行效果如图 20-66 所示。

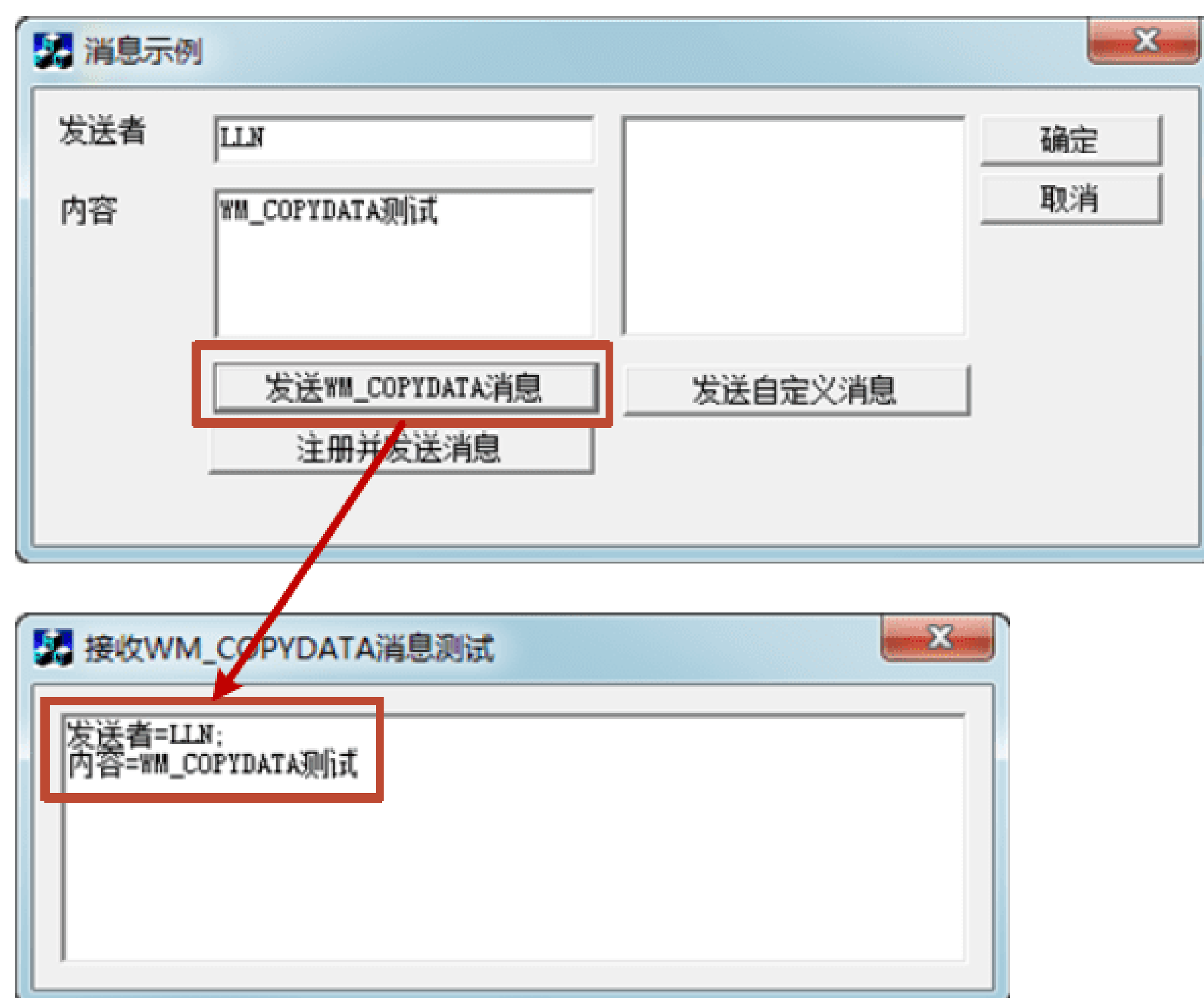


图 20-66　利用 WM_COPYDATA 消息实现进程间数据传递的运行效果

20.9　剪　贴　板

Windows 系统中使用剪贴板实现标准的编辑命令，如剪切命令、复制命令和粘贴命令。而且还可以使用拖曳完成统一的数据传输。在应用程序中，也可以使用剪贴板命令。本节介绍如何使用 Windows API 操作剪贴板。

20.9.1　列举剪贴板中数据类型

调用 EnumClipboardFormats()函数可以枚举当前剪贴板中可用的数据格式。剪贴板数据格式存储在顺序列表中。当使用此函数时需要多次调用，每次调用参数会指定一个可用的剪贴板格式，并且函数会返回下一个有效的剪贴板格式。函数原型为：

```
UINT EnumClipboardFormats(
  UINT format );              //指定已知的可用的剪贴板格式
```

其中参数 format 指定已知的可用的剪贴板格式。当参数设置为 0 时，表示启动剪贴板枚举，函数会获取第一个可用的剪贴板格式。在枚举期间后续的调用，需要设置此参数为前一次调用此函数的返回值。

如果函数执行成功，会返回下一个可用的剪贴板格式。如果函数失败，则会返回 0。剪贴板没有打开和当剪贴板中没有更多的剪贴板格式时，返回值都为 0。此时可以调用 GetLastError()函数判断是哪种情况，如果返回值为 NO_ERROR，表示枚举结束。

在枚举剪贴板数据类型之前，需要使用 OpenClipboard()函数打开剪贴板。此函数会按照放置在剪贴板中的数据枚举出可用的格式。如果将数据复制到剪贴板中，则系统会将其作为剪贴板对象增加到列表中。下面是列举剪贴板中数据类型的代码。

```
01  //列举剪贴板中数据类型
02  void CClicpBoardSampleDlg::OnButtonEnumClformat()
03  {
```

```
04      if(!OpenClipboard())                              //打开剪贴板
05      {
06          WriteLog("打开剪贴板时发生错误");               //输出错误信息
07          return;                                       //返回
08      }
09      WriteLog("剪贴板中支持的数据类型有：");              //输出提示信息
10      UINT uiFormat = EnumClipboardFormats(0);          //开始枚举剪贴板数据类型
11      TCHAR   szName[MAX_PATH];                         //定义剪贴板名称变量
12      while(uiFormat)                               //依次循环处理剪贴板数据格式
13      {
14          if (uiFormat < 0xc000)
15              //如果数据类型小于 0xc000，则调用 GlobalGetAtomName()
16              GlobalGetAtomName((ATOM)uiFormat, szName, MAX_PATH);
17          else                    //否则，调用 GetClipboardFormatName()
18              GetClipboardFormatName(uiFormat, szName, MAX_PATH);
19          WriteLog("%d=%s", uiFormat, szName);      //输出剪贴板格式
20          uiFormat = EnumClipboardFormats(uiFormat);//枚举下一个格式
21      }
22      CloseClipboard();                                 //关闭剪贴板
23  }
```

上面代码调用 OpenClipboard()函数打开剪贴板，并调用 EnumClipboardFormats()函数传入 0，表示开始枚举第一个剪贴板支持的格式，然后使用 while 循环语句依次枚举剪贴板支持的格式，并获取其格式名称。枚举完后，调用 CloseClipboard()函数关闭剪贴板。程序运行效果如图 20-67 所示。

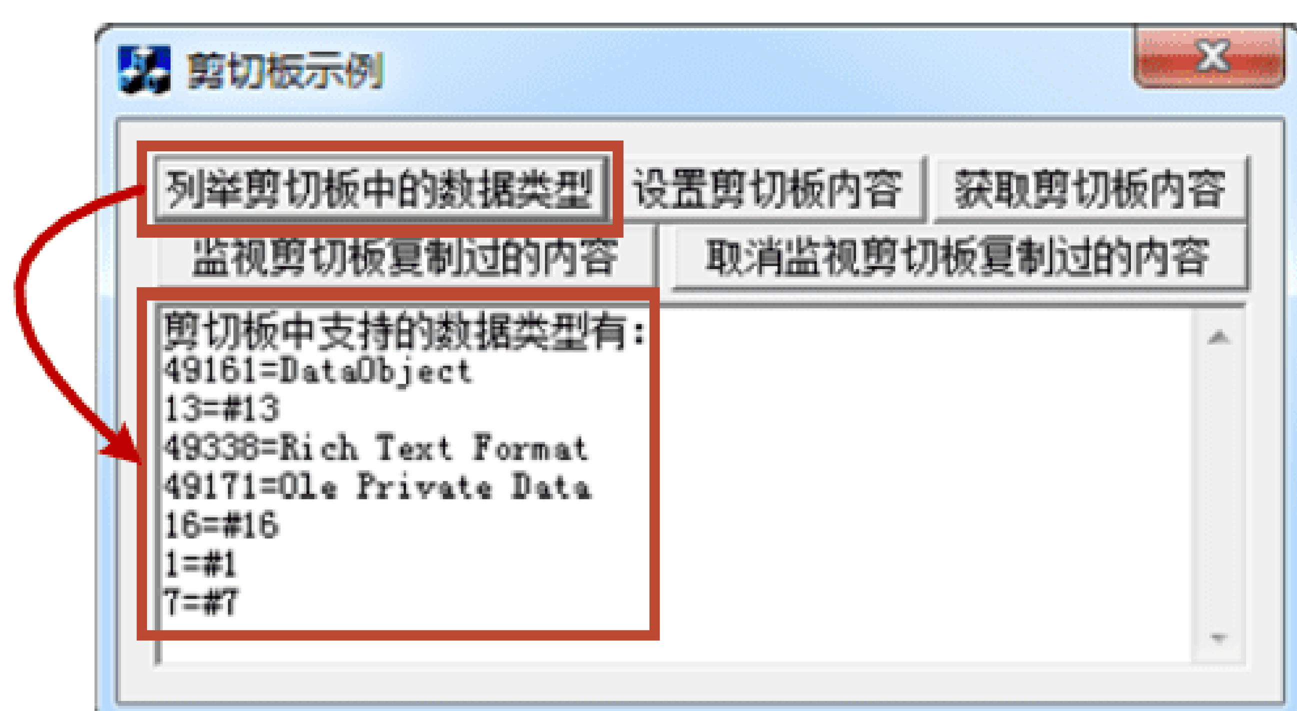


图 20-67　列举剪贴板中的数据类型运行效果

20.9.2　监视剪贴板复制过的内容

要监视剪贴板复制过的内容，需要调用 SetClipboardViewer()函数，将当前对话框添加到剪贴板对话框列表中。然后处理 WM_DRAWCLIPBOARD 消息，此消息当剪贴板内容发生变化时，发送给剪贴板对话框列表中的所有对话框。在监视完剪贴板后，需要调用 ChangeClipboardChain()函数，移除对剪贴板的监视。这两个函数的原型为：

```
HWND SetClipboardViewer(
  HWND hWndNewViewer );                     //要加入剪贴板列表的窗体句柄
BOOL ChangeClipboardChain(
  HWND hWndRemove,                          //要移除剪贴板窗体的窗体句柄
  HWND hWndNewNext );                       //返回的下一个有效窗体句柄
```

以下代码显示了如何启动监视剪贴板的复制功能。

```
01  void CClicpBoardSampleDlg::OnButtonMonitor()//启动监视剪贴板
02  {
03      hNextWnd = SetClipboardViewer();    //将对话框句柄加入剪贴板监视列表
04      if( hNextWnd!= NULL)
05         WriteLog("开始监视剪贴板复制的内容");//输出提示信息
06  }
```

此函数调用 SetClipboardViewer()函数，将当前对话框添加到剪贴板对话框链表中，这样对话框就可以启动监视剪贴板复制内容。当剪贴板的内容发生变化时，会将 WM_DRAWCLIPBOARD 消息发送给剪贴板对话框链表中的所有对话框。消息捕获处理函数代码如下：

```
01  LRESULT CClicpBoardSampleDlg::WindowProc(UINT message,
02            WPARAM wParam, LPARAM lParam)       //消息处理函数
03  {
04      if(message == WM_DRAWCLIPBOARD)           //如果是剪贴板复制消息
05          WriteLog("剪贴板内容发生变化了");       //输出提示信息
06      return CDialog::WindowProc(message, wParam, lParam);
07                                                //调用基类处理函数
08  }
```

上面的程序代码捕获剪贴板内容发生变化的 WM_DRAWCLIPBOAR 消息，收到此消息后，会在日志文本框中显示提示信息。以下代码用来取消监视剪贴板复制过程。

```
01  void CClicpBoardSampleDlg::OnButtonCancelMonitor()//停止监视剪贴板
02  {
03      //将对话框句柄从剪贴板监视列表中移除
04      if( ChangeClipboardChain(hNextWnd))
05      {
06          hNextWnd = NULL;                          //重置监视句柄
07          WriteLog("结束监视剪贴板复制的内容");       //输出提示信息
08      }
09  }
```

上面函数调用 ChangeClipboardChain()函数将当前对话框从剪贴板监视对话框中移除掉。这样，当剪贴板中的内容发生变化时，不再发送 WM_DRAWCLIPBOARD 消息给当前对话框，就结束了对剪贴板复制内容的监视。用户单击“监视剪贴板复制过的内容”按钮后，当剪贴板内容发生变化时，程序会在日志文本器框中输出提示信息。用户单击“取消监视剪贴板复制过的内容”按钮后，当剪贴板内容发生变化时，程序不会收到任何提示信息。程序运行效果如图 20-68 所示。

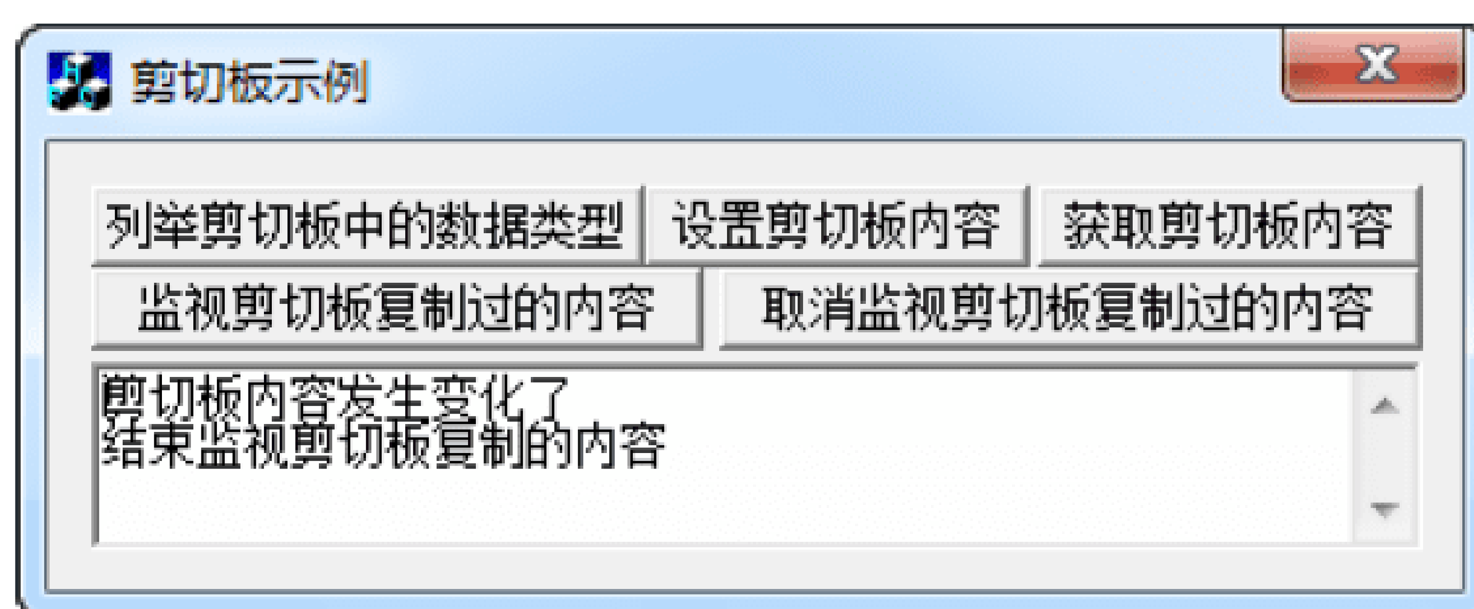


图 20-68　监视剪贴板复制过的内容

20.9.3　通过剪贴板传递全局数据

通过 SetClipboardData()函数和 GetClipboardData()函数可以传递全局数据。其中，SetClipboardData()函数负责将指定的剪贴板格式数据存放到剪贴板中。对话框必须是当前剪贴板的所属对话框，并且应用程序必须已经调用 OpenClipboard()函数打开剪贴板。其函数原型为：

```
HANDLE SetClipboardData(
  UINT uFormat,                    //指定剪贴板格式
  HANDLE hMem);                    //数据句柄
```

在调用此函数时，如果 hMem 参数指定内存对象，则对象必须使用带有 GMEM_MOVEABLE 和 GMEM_DDESHARE 标记的 GlobalAlloc()函数进行分配。如果函数成功，则返回值是数据句柄。

GetClipboardData()函数从剪贴板中获取指定格式的数据。调用此函数前必须调用 OpenClipboard()函数打开剪贴板。其原型为：

```
HANDLE GetClipboardData(
  UINT uFormat );                  //指定了要获取剪贴板数据的格式
```

如果函数成功，则返回指定格式的剪贴板对象的句柄。此句柄由剪贴板而不是应用程序处理，因此应用程序不能长时间使用此句柄。当调用此函数时，系统自动完成隐式的数据格式转换。如果剪贴板中的数据是 CF_OEMTEXT，则对话框可以返回 CF_TEXT 格式的数据。下面代码具体演示了如何通过剪贴板传递全局数据。

```
01  void CClicpBoardSampleDlg::OnButtonSetClicp()       //设置剪贴板数据
02  {
03      LPSTR pBuf = NULL;                              //定义数据区
04      //初始化数据区
05      if (!(pBuf = (LPSTR)GlobalAlloc(GMEM_DDESHARE,
06                                  50 * sizeof(TCHAR))))
07          return;
08      if (!OpenClipboard())                           //打开剪贴板
09      {
10          WriteLog("打开剪贴板时发生错误");              //输出错误信息
11          return;                                     //返回
12      }
13      EmptyClipboard();                               //清空剪贴板
14      CString info;                                   //定义信息提示变量
15      info.Format("通过剪贴板传递全局数据 iIndex=%d", iIndex);//输出提示信息
16      iIndex++;                                       //计数值自增 1
17      strcpy(pBuf, info);
18      if (SetClipboardData(CF_TEXT, pBuf))            //设置剪贴板内容
19          WriteLog("设置剪贴板内容=%s", info);          //输出成功提示信息
20      else
21          WriteLog("设置剪贴板内容是失败");           //输出错误提示信息
22      CloseClipboard();                               //关闭剪贴板
23  }
```

上面的函数将全局变量 iIndex 的值和文字说明组合起来以文本格式存入剪贴板，每次

存入后，会将 index 自增 1。除了文本格式，可以放置任何剪贴板支持的格式的数据到剪贴板中，并且剪贴板可以从中读取任何支持的数据格式。获取剪贴板中的内容的代码如下：

```
01  void CClicpBoardSampleDlg::OnButtonGetClicp()       //获取剪贴板数据
02  {
03      LPSTR pBuf;
04      if (!OpenClipboard())                           //打开剪贴板
05      {
06          WriteLog("打开剪贴板时发生错误");              //输出错误信息
07          return;                                     //返回
08      }
09      //获取剪贴板中的 CF_TEXT 数据
10      HGLOBAL hGlobal = GetClipboardData(CF_TEXT);
11      pBuf = (LPSTR)GlobalLock(hGlobal);              //锁定数据区
12      WriteLog("获取剪贴板内容=%s", pBuf);           //输出获取的剪贴板内容
13      GlobalUnlock(hGlobal);                          //解锁数据区
14      CloseClipboard();                               //关闭剪贴板
15  }
```

上面代码打开剪贴板，然后调用 GetClipboardData()函数获取文本格式的剪贴板数据，调用 GlobalLock()函数锁定内存句柄，从中取出数据，并显示在界面上，最后解锁内存句柄并关闭剪贴板。程序运行后，当用户单击“设置剪贴板内容”按钮时，程序会将当前 index 值和文本组合放入剪贴板。当用户单击“获取剪贴板内容”按钮时，程序会在日志文本框中显示当前剪贴板中的文本内容。程序运行效果如图 20-69 所示。

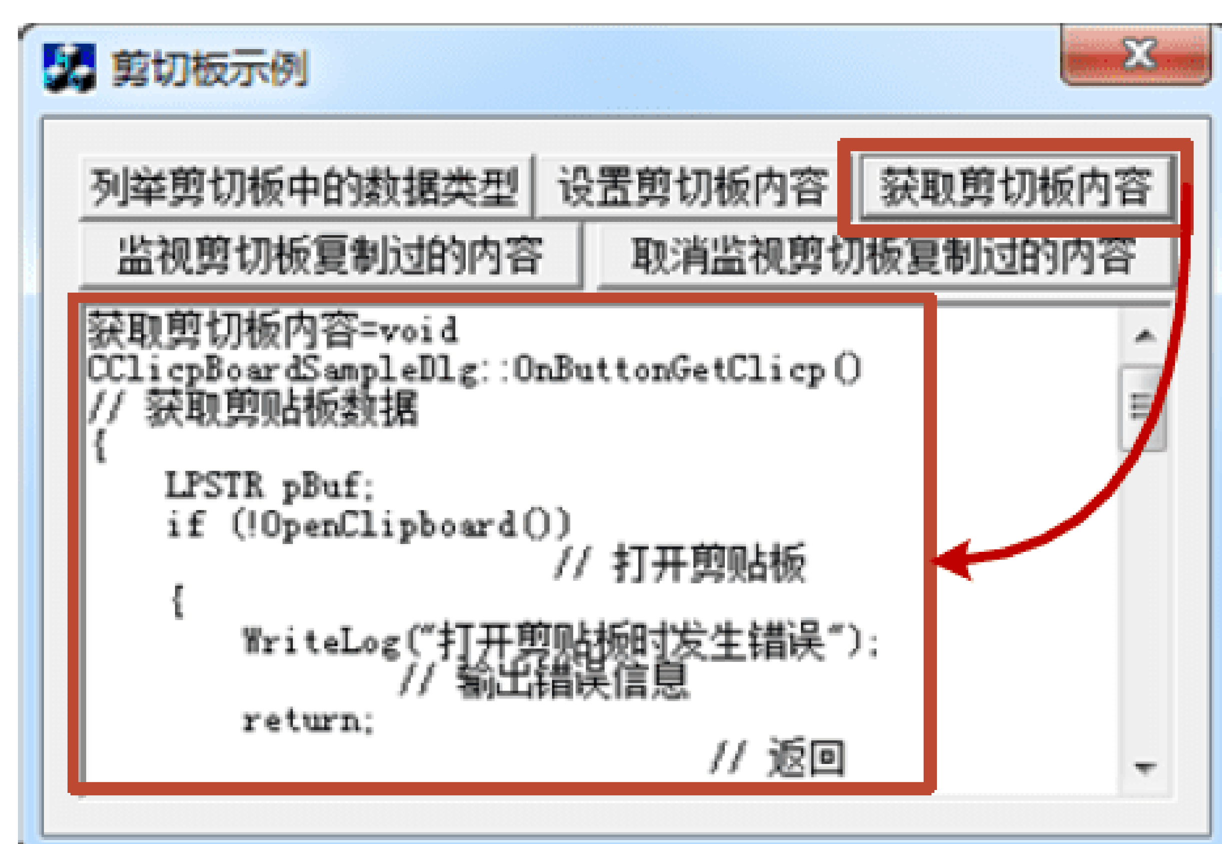


图 20-69　通过剪贴板传递全局数据的运行效果

20.10　鼠 标 键 盘

用户最常用的输入设备就是鼠标和键盘。因此，要开发界面友好的程序，很多时候需要对鼠标键盘做特殊处理，本节介绍有关鼠标键盘的操作。如交换鼠标左右键、设置双击间隔时间、获取鼠标按键数、模拟鼠标事件、添加加速键、处理鼠标滚轮消息、获取键盘信息以及控制键盘指示灯等方面的知识。

20.10.1　交换鼠标左右键

在实际操作中有些人习惯使用左手，系统为其提供了 SwapMouseButton()函数，可以

交换鼠标左键和鼠标右键。虽然应用程序可以随意地调用此函数，但通常情况下，此函数只能由控制面板调用，因为鼠标是个共享资源，交换左右键后，会影响到系统中的所有程序，所以要谨慎使用。其函数原型为：

```
BOOL SwapMouseButton(
  BOOL fSwap);            //表示是要交换鼠标左右键还是恢复默认的左右键设置
```

上面的函数中 fSwap 参数为 true，则左键产生鼠标右键消息，鼠标右键产生鼠标左键消息。如果参数为 false，则恢复默认的鼠标左右键功能。此函数的返回值的含义比较特殊。如果调用此函数前，鼠标左右键是交换的，则返回非 0 值（true）。如果没有交换鼠标左右键，则返回 0（false）。代码如下：

```
01  void CMouseKeyBordSampleDlg::OnButSwap()    //交换鼠标左右键
02  {
03      if (SwapMouseButton(true))              //交换鼠标左右键
04          WriteLog("鼠标左右键已经交换");        //输出错误提示信息
05      else
06          WriteLog("交换鼠标左右键成功");        //输出提示信息
07  }
08  void CMouseKeyBordSampleDlg::OnButRestoreswap()//恢复鼠标左右键
09  {
10      if (SwapMouseButton(false))             //恢复鼠标左右键
11          WriteLog("恢复鼠标左右键成功");        //输出提示信息
12      else
13          WriteLog("交换鼠标左右键成功");        //输出提示信息
14  }
```

上面代码中显示了交换鼠标左右键函数的使用。当交换鼠标左右键时，返回的是 false。当恢复鼠标左右键时，返回的是 true。程序运行效果如图 20-70 所示。

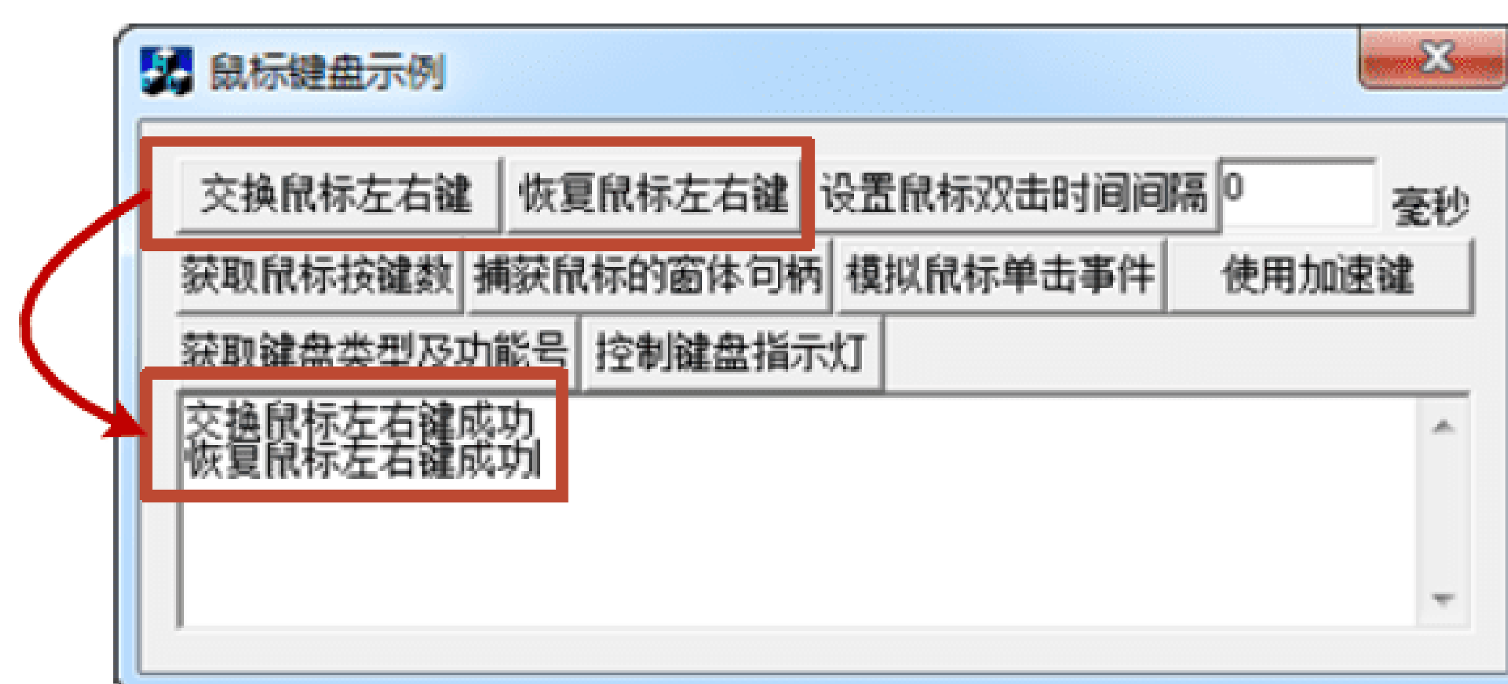


图 20-70　交换鼠标左右键运行效果

20.10.2　设置鼠标双击的时间间隔

双击即鼠标按钮连续按下两次的事件，第二次按下时，必须在第一次按下后指定时间内按下才算双击，否则，系统会作为两次单击事件处理。鼠标双击的时间间隔是双击事件两次按下鼠标之间的最大时间间隔，使用 SetDoubleClickTime()函数可以设置其值。函数原型为：

```
BOOL SetDoubleClickTime(
  UINT uInterval);              //指定鼠标双击的时间间隔，单位是毫秒
```

如果 uInterval 参数设置为 0，则系统使用默认的双击时间间隔 500 毫秒。函数设置的双击时间间隔应用于系统中的所有对话框。使用 GetDoubleClickTime()函数可以获取当前设置的鼠标双击时间间隔。函数原型为：

```
UINT GetDoubleClickTime(VOID)
```

此函数没有参数，返回值为当前设置的双击时间间隔，单位是毫秒。以下代码显示了设置和获取鼠标双击时间间隔的函数的调用方法。

```
01  void CMouseKeyBordSampleDlg::OnButSetdoubletime()
02  {
03      UpdateData(true);                       //获取要设置的鼠标双击时间间隔
04      //设置双击时间间隔值
05      BOOL bResult = SetDoubleClickTime(m_DoubleClickTime);
06      UINT dcTime = GetDoubleClickTime(); //获取双击时间间隔值
07      WriteLog("设置鼠标双击时间间隔%s，当前值为%d",
08                  bResult ? "成功": "失败",dcTime);
09  }
```

上面代码是设置鼠标双击时间间隔的按钮的单击处理事件，从控件中更新用户输入的值到变量 m_DoubleClickTime 中，然后调用 SetDoubleClickTime()函数设置，并调用 GetDoubleClickTime()函数获取设置后的值，最后在日志文本框中提示用户设置结果。程序运行结果如图 20-71 所示。

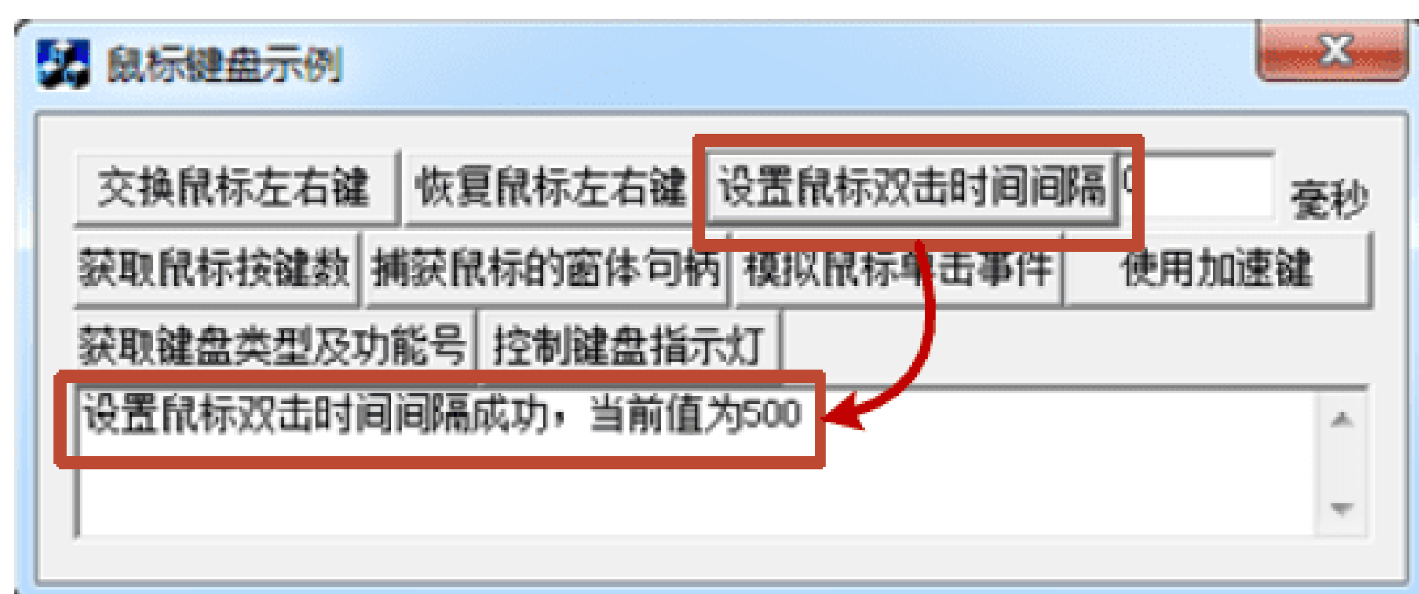


图 20-71　设置鼠标双击时间间隔运行效果

20.10.3　获得鼠标键数

前面讲过获取系统信息的 GetSystemMetrics()函数，传入 SM_CMOUSEBUTTONS 参数调用此函数，可以获得鼠标键数。代码如下：

```
01  void CMouseKeyBordSampleDlg::OnButtonGetbuttons()      //获得鼠标键数
02  {
03      int iButtons = GetSystemMetrics(SM_CMOUSEBUTTONS); //获取按键数
04      if (iButtons > 0)                                  //如果大于 0
05          WriteLog("当前系统中共有%d 个鼠标按键", iButtons);  //输出按键数
06      else
07          WriteLog("当前系统中没有安装鼠标");      //否则，系统中没有安装鼠标
08  }
```

上面代码调用 GetSystemMetrics()函数传入 SM_CMOUSEBUTTONS 参数获得鼠标键数，如果返回值为 0，表示系统中没有安装鼠标，否则返回值为系统中的鼠标按键数。程序运行效果如图 20-72 所示。

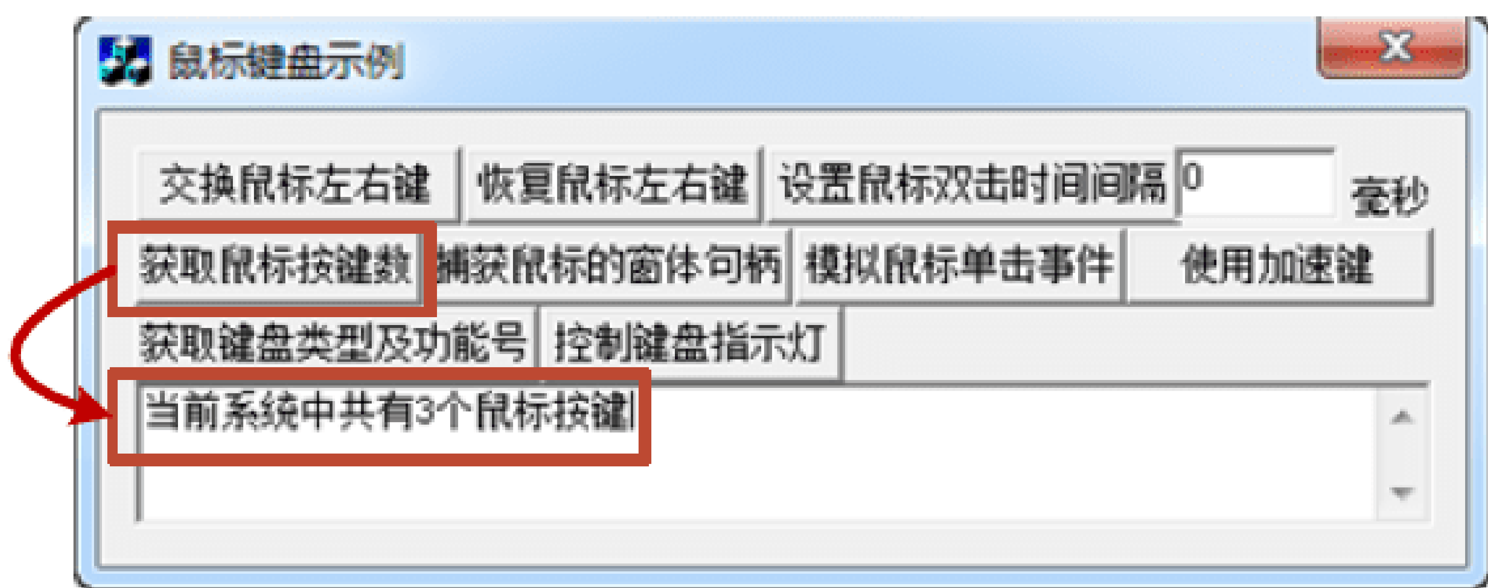


图 20-72　获得鼠标按键数

20.10.4　获取鼠标下窗体句柄

调用 GetCapture()函数可以获取捕获鼠标的对话框句柄。同一时间鼠标只能捕获一个对话框。无论光标是否在对话框的边框内，对话框接收鼠标的输入。其函数原型为：

```
HWND GetCapture(VOID)
```

此函数没有参数，返回值为与当前线程相连的捕获的对话框的句柄。如果当前线程没有对话框捕获鼠标，则返回值为 NULL。下面的代码演示了如何获取鼠标下窗体句柄。

```
01  void CMouseKeyBordSampleDlg::OnTimer(UINT nIDEvent)//定时器处理函数
02  {
03      HWND m_hWnd  = ::GetCapture();                       //捕获鼠标窗体
04      WriteLog("当前捕获鼠标的窗体的 HWND=0x%08X", m_hWnd);//输出句柄
05      CDialog::OnTimer(nIDEvent);                          //调用基类处理函数
06  }
```

上面代码调用 GetCapture()函数捕获鼠标下窗体句柄，并将结果显示在日志文本框中。程序运行效果如图 20-73 所示。

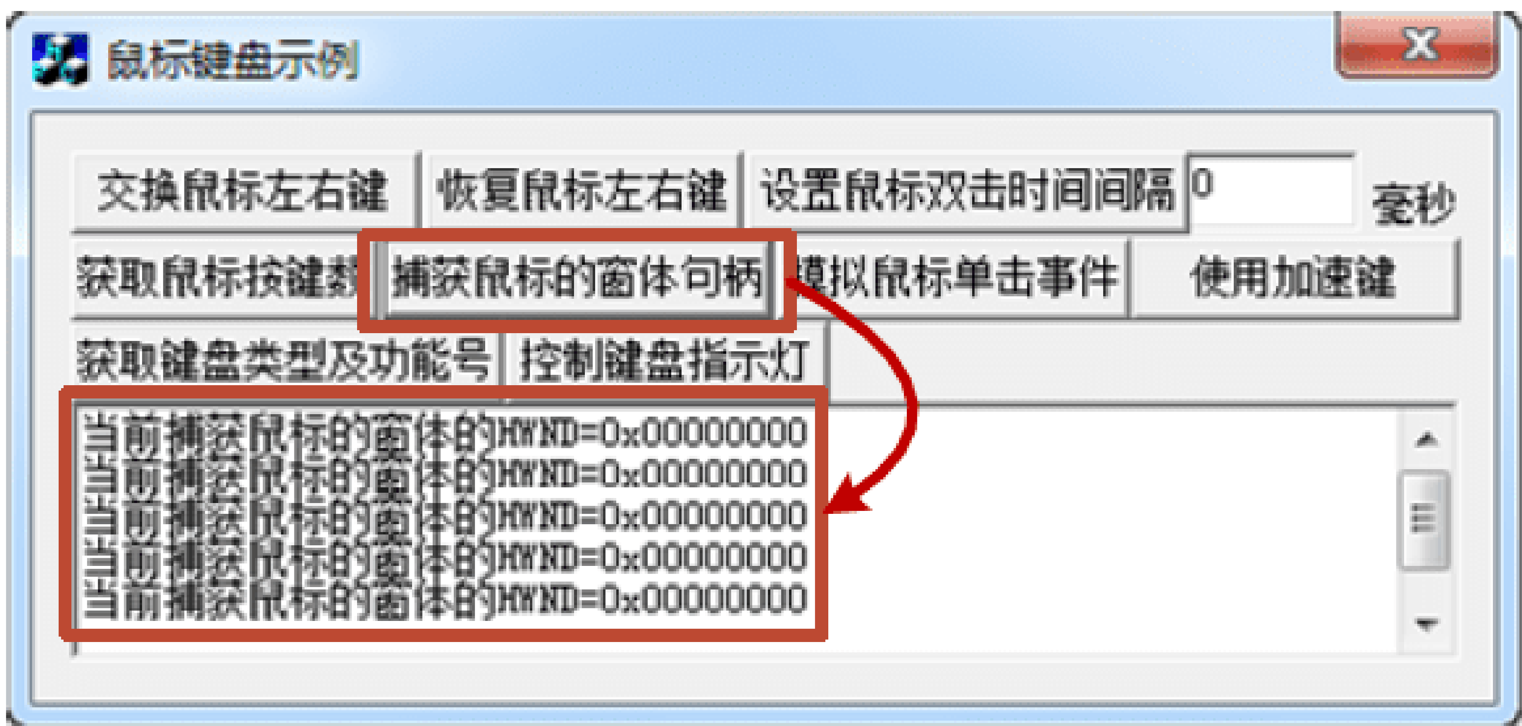


图 20-73　捕获鼠标的窗体句柄运行效果

20.10.5　模拟鼠标单击按钮

调用 mouse_event()函数可以模拟鼠标和按钮的单击事件。

```
VOID mouse_event(
  DWORD dwFlags,                      //指定要模拟的鼠标和按键行为的种类
  DWORD dx,                           //指定鼠标位置或位置变化的水平坐标
  DWORD dy,                           //指定鼠标位置或位置变化的垂直坐标
```

```
  DWORD dwData,                     //指定鼠标滚轮移动的距离
  DWORD dwExtraInfo);               //存放通过事件传递的信息
```

接收事件的函数中调用 GetMessageExtraInfo()函数可以获取 dwExtraInfo 参数传递的信息。表 20-10 列出了函数支持的鼠标事件种类。

表 20-10　mouse_event()函数支持的鼠标事件

值	含　义
MOUSEEVENTF_ABSOLUTE	指定 dx 和 dy 参数中指定的是绝对坐标
MOUSEEVENTF_MOVE	指定鼠标移动事件
MOUSEEVENTF_LEFTDOWN	指定左键按下事件
MOUSEEVENTF_LEFTUP	指定左键抬起事件
MOUSEEVENTF_RIGHTDOWN	指定右键按下事件
MOUSEEVENTF_RIGHTUP	指定右键抬起事件
MOUSEEVENTF_MIDDLEDOWN	指定中键按下事件
MOUSEEVENTF_MIDDLEUP	指定中键抬起事件
MOUSEEVENTF_WHEEL	指定滚轮移动事件

下面代码显示了如何模拟鼠标单击事件，会在鼠标单击的原处模拟一次鼠标单击事件，并且在消息处理函数中，捕获左键按下和左键抬起事件。

```
01  LRESULT CMouseKeyBordSampleDlg::WindowProc(UINT message,
02      WPARAM wParam, LPARAM lParam)   //捕获消息
03  {
04      //捕获左键按下消息
05      if (message == WM_LBUTTONDOWN)
06          WriteLog("按下鼠标左键");
07      //捕获左键抬起消息
08      else if (message == WM_LBUTTONUP)
09          WriteLog ("抬起鼠标左键");
10      return CDialog::WindowProc(message, wParam, lParam);
11  }
12  void CMouseKeyBordSampleDlg::OnButSendmouse()//模拟鼠标左键
13  {
14      mouse_event(MOUSEEVENTF_LEFTDOWN | MOUSEEVENTF_LEFTUP ,
15          0, 0, NULL,NULL);
16  }
```

当用户单击“模拟鼠标单击事件”按钮时，程序会在鼠标单击的地方重新触发一次鼠标单击事件，程序捕获后，在日志文本框中显示提示信息。程序运行效果如图 20-74 所示。

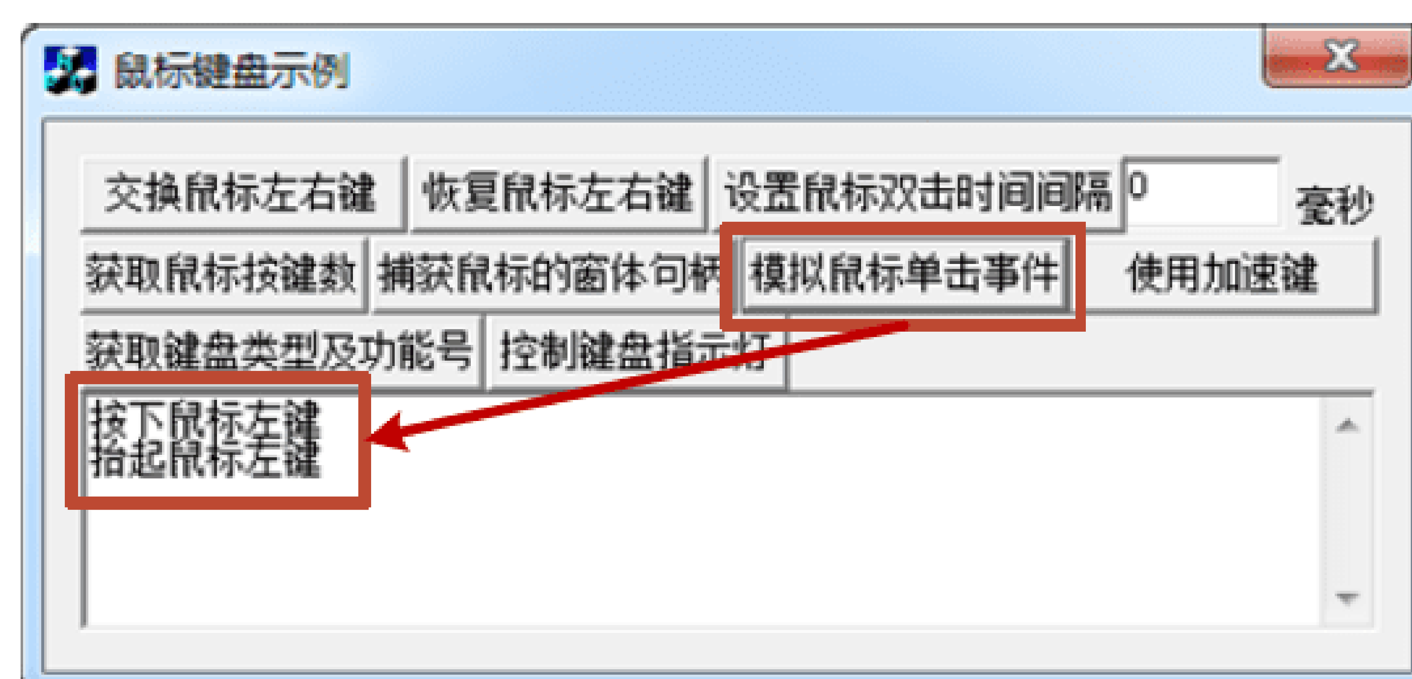


图 20-74　模拟鼠标单击按钮的运行效果

20.10.6　在程序中添加快捷键

快捷键是程序中为了方便用户而设置的可以快速执行相应任务的按键或按键组合，除了在菜单项或按钮的文本后加上"(&快捷键按键)"的方式外，还可以在程序中通过在PreTranslateMessage()函数中过滤要设置的快捷键为程序添加快捷键。方法是调用GetAsyncKeyState()函数判断是否有要处理的快捷键被按下。其函数原型为：

```
SHORT GetAsyncKeyState( int vKey);      //要判断的快捷键
```

函数返回值如果大于 0，表示查询的快捷键被按下了。代码如下：

```
01  BOOL CMouseKeyBordSampleDlg::PreTranslateMessage(MSG* pMsg)
02  {
03      if(GetAsyncKeyState(VK_F5))                 //判断是否是所要设置的快捷键
04      {
05          OnButtonGetbuttons();                   //执行快捷键要处理的函数
06          return  true;                           //返回
07      }
08      return CDialog::PreTranslateMessage(pMsg);
09  }
```

上面代码添加了 OnButtonGetbuttons()函数的快捷键 F5 键，当用户按下 F5 键后，系统会自动执行获取鼠标按键数的函数。程序运行效果，如图 20-75 所示。

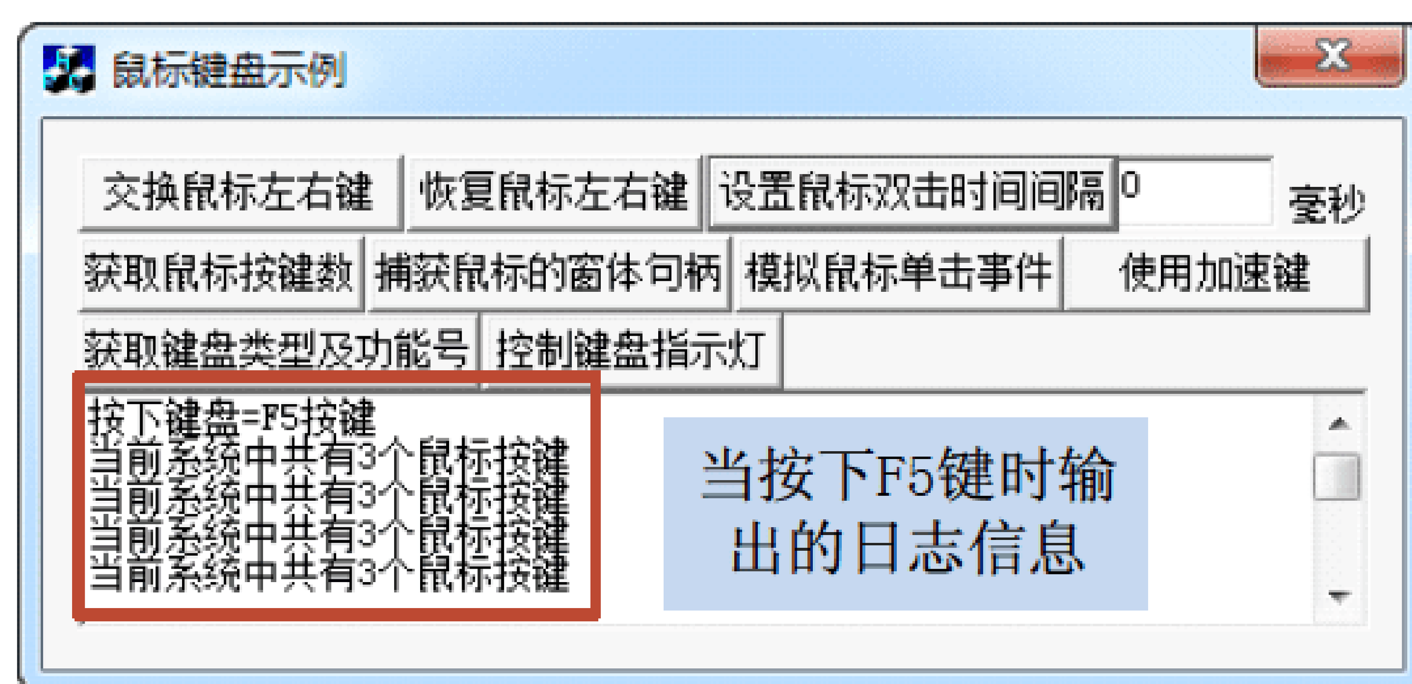


图 20-75　添加快捷键运行效果

20.10.7　在对话框中使用加速键

加速键就是在程序中指定的具有特殊功能的按键组合，读者可以根据自己的需要向系统注册完成特殊作用的加速键。使用 RegisterHotKey()函数可以定义系统级别的加速键。其函数原型为：

```
BOOL RegisterHotKey(
  HWND hWnd,                  //表示接收加速键 WM_HOTKEY 消息的窗体的句柄
  int id,                     //指定加速键的标识符
  UINT fsModifiers,           //指定与指定键组合在一起的功能键
  UINT vk );                  //指定加速键的虚拟键值
```

fsModifiers 参数指定与指定键组合在一起的功能键，其有效取值有 MOD_ALT、MOD_CONTROL、MOD_SHIFT 和 MOD_WIN，分别表示按下加速键时，需要同时按下

Alt 键、Ctrl 键、Shift 键和 Win 键。下面代码显示了使用加速键的方法。

```
01  void CMouseKeyBordSampleDlg::OnButRegisthotkey()
02  {
03      if(RegisterHotKey(this|m_hWnd, VK_F1+1, MOD_CONTROL, VK_F1+1))
04                                                              //注册加速键
05          WriteLog("注册加速键成功");                         //输出提示信息
06      else
07          WriteLog("注册加速键失败");                         //输出错误信息
08  }
```

上面的代码使用 RegisterHotKey()函数向系统注册了 F2+Ctrl 组合的加速键。注册完加速键后，读者可以通过捕获 WM_HOTKEY 消息处理加速键的使用。代码如下：

```
01  LRESULT CMouseKeyBordSampleDlg::WindowProc(UINT message,
02      WPARAM wParam, LPARAM lParam)                    //窗口消息处理函数
03  {
04      char szName[MAX_PATH]={0};                       //定义名称变量
05      UINT fuModifiers,uVirtKey;                       //定义按键变量
06      switch(message)                                  //判断消息
07      {
08          ...//此处代码省略
09      case WM_HOTKEY:                                  //接收到加速键
10          //获取功能按键
11          uVirtKey = MapVirtualKey((UINT)HIWORD(lParam), 0)<<16;
12          //获取按键名称
13          if (GetKeyNameText(uVirtKey, (LPTSTR)szName, sizeof(szName)))
14              WriteLog ("\r\n 加速键=%s", szName);    //输出按键名称
15          fuModifiers = (UINT) LOWORD(lParam);        //取组合功能键
16          switch(fuModifiers)                          //判断组合键
17          {
18          case MOD_Alt:                                //Alt 键
19              WriteLog("（按下 Alt 组合键）");
20              break;
21          case MOD_CONTROL:                            //Ctrl 键
22              WriteLog("（按下 Ctrl 组合键）");
23              break;
24          case MOD_Shift:                              //Shift 键
25              WriteLog("（按下 Shift 组合键）");
26              break;
27          case MOD_WIN:                                //Win 键
28              WriteLog("（按下 Win 组合键）");
29              break;
30          }
31          break;
32          ...                                          //此处代码省略
33      }
34      return CDialog::WindowProc(message, wParam, lParam);
35  }
```

上面代码捕获了 WM_HOTKEY 消息，取出是否按下组合键以及按下的按键的虚拟键值，并在日志文本框中显示出来。读者可以根据需要对捕获的加速键进行处理。程序运行效果如图 20-76 所示。

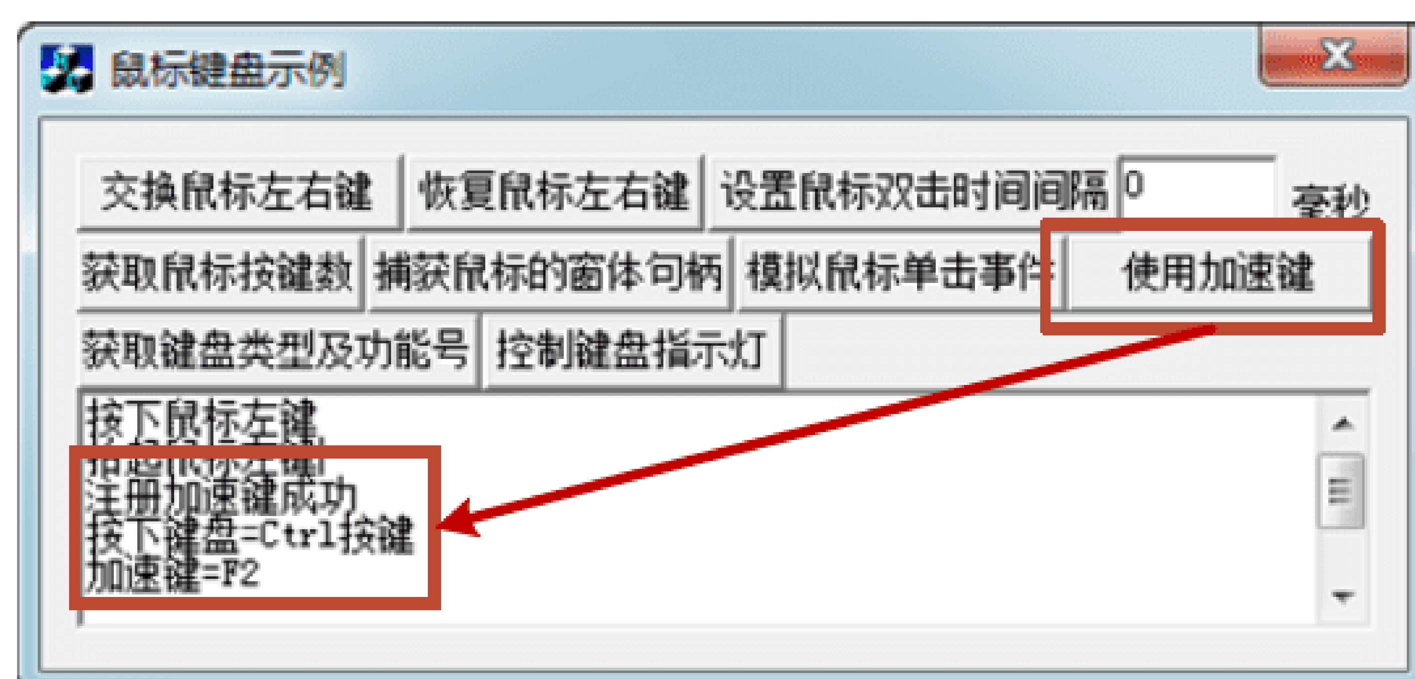


图 20-76　在对话框中使用加速键的运行效果

20.10.8　处理鼠标滚轮消息

Windows 中标识鼠标滚轮的消息是 WM_MOUSEWHEEL，当鼠标滚轮滚动时，系统会发送此消息到当前的焦点对话框中。其发送的参数如下：

```
fwKeys = LOWORD(wParam);              //表示滚动滚轮时同时按下的按键
zDelta = (short) HIWORD(wParam);      //滚轮移动的值
xPos = (short) LOWORD(lParam);        //当前鼠标的水平位置点
yPos = (short) HIWORD(lParam);        //当前鼠标的垂直位置点
```

其中 zDelta 参数表示滚轮滚动的值，是 WHEEL_DELTA 值的整数倍，WHEEL_DELTA 的值为 120。如果 zDelta 参数为正，表示滚轮向前滚动，如果 zDelta 参数为负，表示滚轮向后滚动。在 MFC 程序中，提供了默认的处理滚轮事件的 OnMouseWheel()函数。

```
afx_msg BOOL OnMouseWheel(
  UINT nFlags,                    //消息中的 fwKeys 值
  short zDelta,                   //消息中的 zDelta 值
  CPoint pt );                    //存储消息中的 xPos 和 yPos 的值
```

下面的代码演示了滚轮消息的处理。

```
01  //处理鼠标滚轮消息
02  BOOL CMouseKeyBordSampleDlg::OnMouseWheel(UINT nFlags, short zDelta,
03                                            CPoint pt)
04  {
05      WriteLog("发生鼠标滚轮事件--");          //输出提示信息
06      switch(nFlags)                          //判断有没有按下组合键，并输出
07      {
08      case MK_CONTROL:
09          WriteLog("Ctrl 按键按下。");
10          break;
11      case MK_LBUTTON:
12          WriteLog("鼠标左键按下。");
13          break;
14      case MK_MBUTTON:
15          WriteLog("鼠标中键按下。");
16          break;
17      case MK_RBUTTON:
18          WriteLog("鼠标右键按下。");
19          break;
20      case MK_SHIFT:
21          WriteLog("Shift 按键按下。");
```

```
22            break;
23        default:
24            WriteLog("没有功能键按下。");
25            break;
26        }
27        if (zDelta > 0)                          //输出滚轮移动的方向和大小
28            WriteLog("滚轮向前移动%d", abs(zDelta));
29        else
30            WriteLog("滚轮向后移动%d", abs(zDelta));
31        WriteLog("鼠标当前位置(相对于屏幕左上角), X=%d;Y=%d",pt.x, pt.y);
32        //输出鼠标当前位置
33        return CDialog::OnMouseWheel(nFlags, zDelta, pt);
34    }
```

上面代码使用 switch 语句判断是否有其他按键被同时按下，然后判断滚轮滚动了多少，是向前滚动还是向后滚动，最后获取了鼠标当前的位置，并在日志文本框中显示这些信息。程序运行效果如图 20-77 所示。

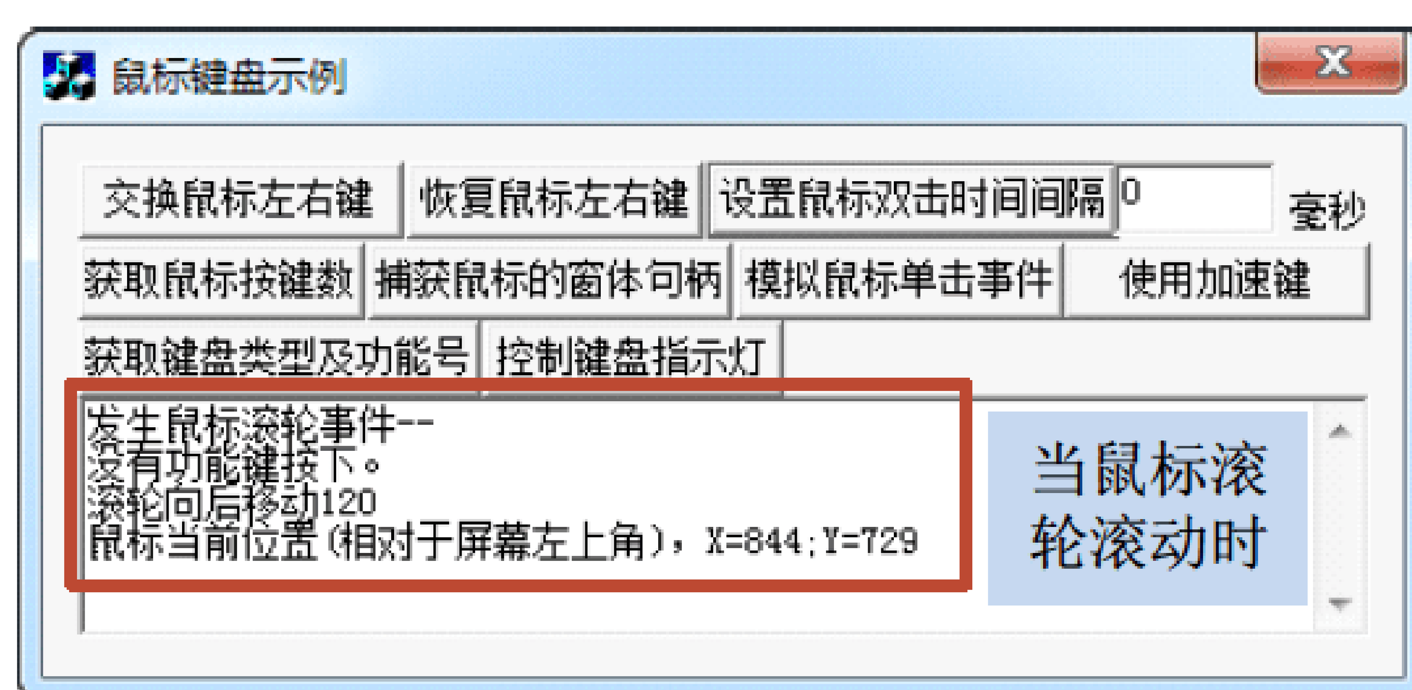


图 20-77　处理鼠标滚轮消息的运行效果

20.10.9　获取键盘按键

在消息预处理函数中，用户可以通过捕获 WM_KEYDOWN 消息获取键盘按键的消息，并通过 GetKeyNameText()函数获取按下的按键名称。

```
nVirtKey = (int) wParam;   lKeyData = lParam;   //WM_KEYDOWN 消息参数
```

从上面可以看出，wParam 参数是按下的键盘按键的虚拟键值，lKeyData 参数表示按键的次数、上下文代码等其他信息，一般不需要处理。GetKeyNameText()函数的原型为：

```
int GetKeyNameText(
  LONG lParam,                         //lParam 参数
  LPTSTR lpString,                     //存放按键名称的地址
  int nSize);                          //表示存储区的大小
```

此函数获取了指定键值的按键名称，如果函数返回值不为 0，则表示成功地获取了按键的名称。以下代码显示了如何获取键盘按键。

```
01  //消息预处理
02  BOOL CMouseKeyBordSampleDlg::PreTranslateMessage(MSG* pMsg)
03  {
04      if (pMsg->message == WM_KEYDOWN)             //判断是否是按键按下
05      {
06          char szName[MAX_PATH]={0};               //定义按键名称数组
```

```
07          //获取按键值
08          UINT uVirtKey = MapVirtualKey(pMsg->wParam, 0)<<16;
09          //获取按键名称
10          if (GetKeyNameText(uVirtKey, (LPTSTR)szName, sizeof(szName)))
11              WriteLog("按下键盘=%s 按键", szName);   //输出按键名称
12      }
13      return CDialog::PreTranslateMessage(pMsg); //基类处理函数
14  }
```

上面代码判断消息是否为 WM_KEYDOWN，如果是，则调用 GetKeyNameText()函数获取按键的中文名称。需要注意的是，在将按键的虚拟键值传入 GetKeyNameText()函数前，需要调用 MapVirtualKey()函数将消息传递的按键值映射成虚拟键值。当在键盘上按下按键时，在日志文本框中会显示出按下的键盘按键名称。程序运行效果如图 20-78 所示。

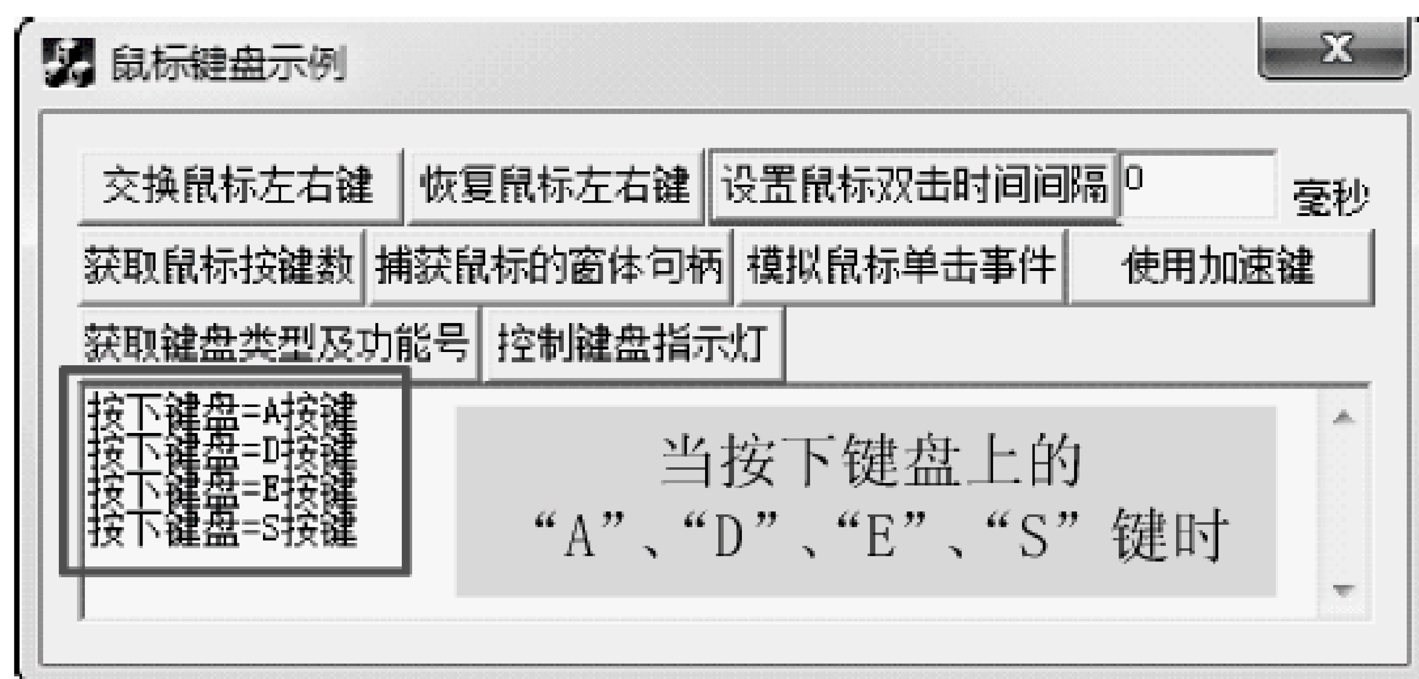


图 20-78　获取键盘按键运行效果

20.10.10　获取键盘类型及功能号

调用 GetKeyboardType()函数可以获取当前键盘信息。其函数原型为：

```
int GetKeyboardType(
  int nTypeFlag );       //指定要获取的键盘信息的类型
```

参数 nTypeFlag 用于指定要获取的键盘信息的类型。有效取值为 0、1、2，分别表示键盘类型、键盘子类型和键盘上的功能键个数。表 20-11 和表 20-12 分别列出了键盘类型和键盘功能键数目的有效返回值。

表 20-11　键盘类型有效返回值

值	含　义	值	含　义
1	IBM PC/XT 或兼容（83 键）键盘	5	Nokia 1050 系列键盘
2	Olivetti ICO（102 键）键盘	6	Nokia 9140 系列键盘
3	IBM PC/AT （84 键）系列键盘	7	Japanese 键盘
4	IBM enhanced （101 或 102 键）键盘		

表 20-12　键盘功能键个数返回值

键 盘 类 型	功能键个数	键 盘 类 型	功能键个数
1	10	5	10
2	12（有时 18）	6	24
3	10	7	根据 OEM 厂商而不同
4	12		

下面的代码演示了如何获取键盘类型及功能键数目。

```
01  void CMouseKeyBordSampleDlg::OnButtonGetkeyboard()//获取键盘类型
02  {
03      //定义键盘类型名称
04      CString type[8]={"获取键盘类型失败",
05          "IBM PC/XT or compatible (83-key)keyboard",
06          "Olivetti 'ICO' (102-key) keyboard",
07          "IBM PC/AT (84-key) or similar  keyboard",
08          "IBM enhanced (101- or 102-key) keyboard",
09          "Nokia 1050 and similar keyboards",
10          "Nokia 9140 and similar keyboards",
11          "Japanese keyboard"};
12      CString funKeys[8]={"获取键盘功能键数目失败", "10",
13          "12,有时是 18","10",  "12","10", "24",
14          "根据 OEM 厂商而定"};                        //定义键盘功能键数目
15      int keyboardType = GetKeyboardType(0);       //获取键盘类型
16      int keyboardSubType = GetKeyboardType(1);    //获取键盘子类型
17      int functionsKeys = GetKeyboardType(2);      //获取键盘功能键数目
18      if (keyboardType < 8)
19          WriteLog("键盘类型=%s", type[keyboardType]);//输出键盘类型
20      else
21          WriteLog("键盘类型未知");
22      //输出键盘子类型
23      WriteLog("键盘子类型=%d(具体含义,
24                  需要根据键盘类型的不同进行判断)",keyboardSubType);
25      //输出键盘功能键数目
26      if (functionsKeys < 8)
27          WriteLog("键盘功能键数目=%s\r\n",type[functionsKeys]);
28      else
29          WriteLog("键盘功能键数目未知");
30  }
```

上面代码调用 GetKeyboardType()函数分别获取了键盘类型、键盘子类型和键盘上功能键的个数，并根据返回值，将含义显示在日志文本框中。程序运行效果如图 20-79 所示。

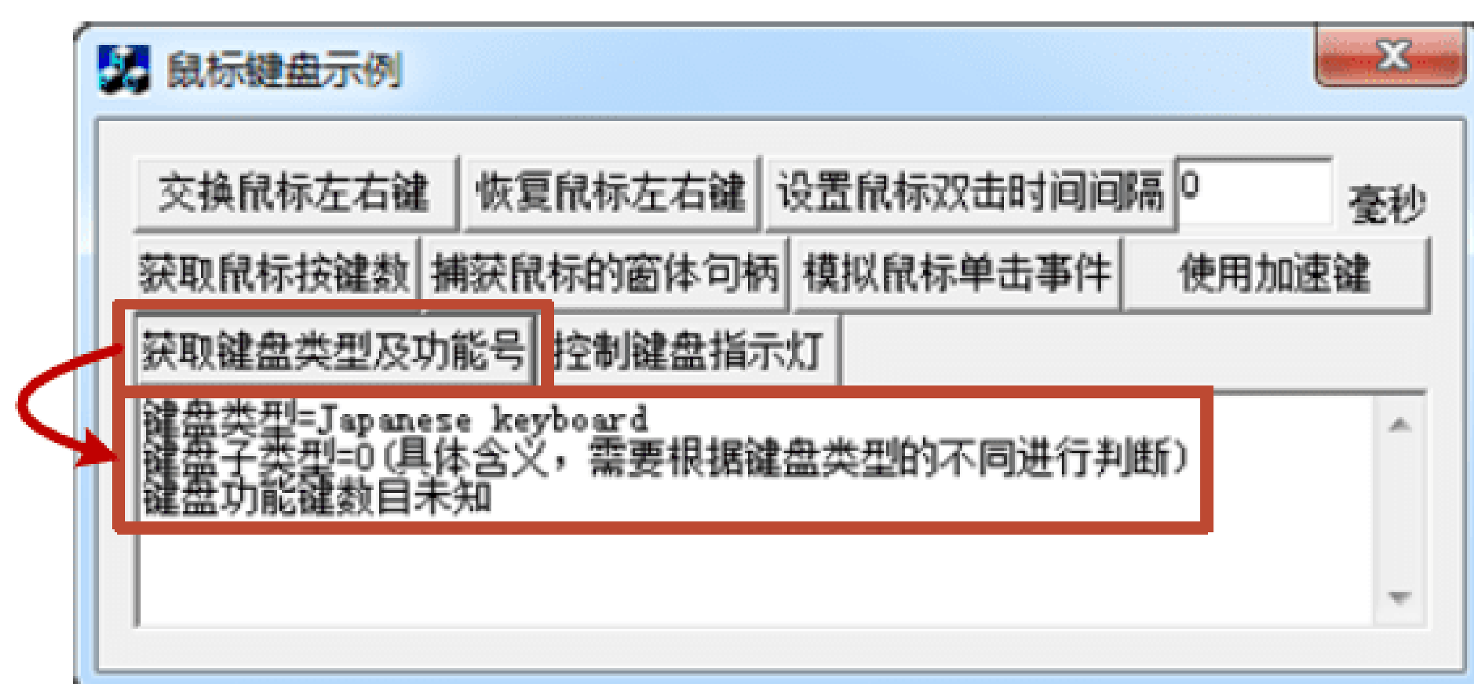


图 20-79　获取键盘类型及功能号的运行效果

20.10.11　控制键盘指示灯

调用 keybd_event()函数可以控制键盘的按键事件，通过模拟键盘按键事件，可以控制

键盘指示灯的状态。本小节讲解如何控制大小写切换指示灯、数字键盘键指示灯和滚动键指示灯的状态。keybd_event()函数原型为：

```
VOID keybd_event(
  BYTE bVk,                     //指定要发送键盘事件的虚拟按键代码，取值为 1～254
  BYTE bScan,                   //指定按键的硬件扫描码
  DWORD dwFlags,                //操作选项
  DWORD dwExtraInfo);           //指定通过按键事件传递的数据
```

其中 dwFlags 参数表示操作选项，如果指定 KEYEVENTF_EXTENDEDKEY 选项，则系统会对扫描码的值进行处理，如果指定 KEYEVENTF_KEYUP 选项，则按键事件发生后会释放。以下代码显示了如何调用 keybd_event()函数控制键盘指示灯。

```
01  void CMouseKeyBordSampleDlg::OnButControlkey()
02  {
03      //定义控制的指示灯
04      int keys[] = {VK_CAPITAL, VK_NUMLOCK,VK_SCROLL};
05      //定义指示灯名称
06      CString keysName[] = {"VK_CAPITAL", "VK_NUMLOCK",
07          "VK_SCROLL"};
08      //依次控制各个指示灯状态
09      for (int i = 0;i < sizeof(keys)/sizeof(int);i ++)
10      {
11          //切换指示灯状态
12          keybd_event( keys[i], 0x45, KEYEVENTF_EXTENDEDKEY
13              | 0, 0 );
14          keybd_event( keys[i], 0x45, KEYEVENTF_EXTENDEDKEY
15              | KEYEVENTF_KEYUP, 0);
16      }
17      BYTE keyState[MAX_PATH]={0};                  //指示灯状态数组
18      GetKeyboardState((LPBYTE)&keyState);          //获取键盘状态
19      //依次获取指示灯状态并输出
20      for (i = 0;i < sizeof(keys)/sizeof(int);i ++)
21          WriteLog("%s 当前%s",keysName[i],
22          keyState[keys[i]] == 1 ? "灯灭":"灯亮");
23  }
```

上面代码定义了与要控制的 3 个指示灯关联的按键值和按键名称。对这 3 个按键依次调用 keybd_event 事件，则指示灯的状态会发生改变，最后调用 GetKeyboardState()函数获取当前按键的状态，并通过要处理的按键的键值作为索引从数组中读取当前按键的状态。当用户单击“控制键盘指示灯”按钮时，系统会切换指示灯状态。程序运行效果如图 20-80 所示。

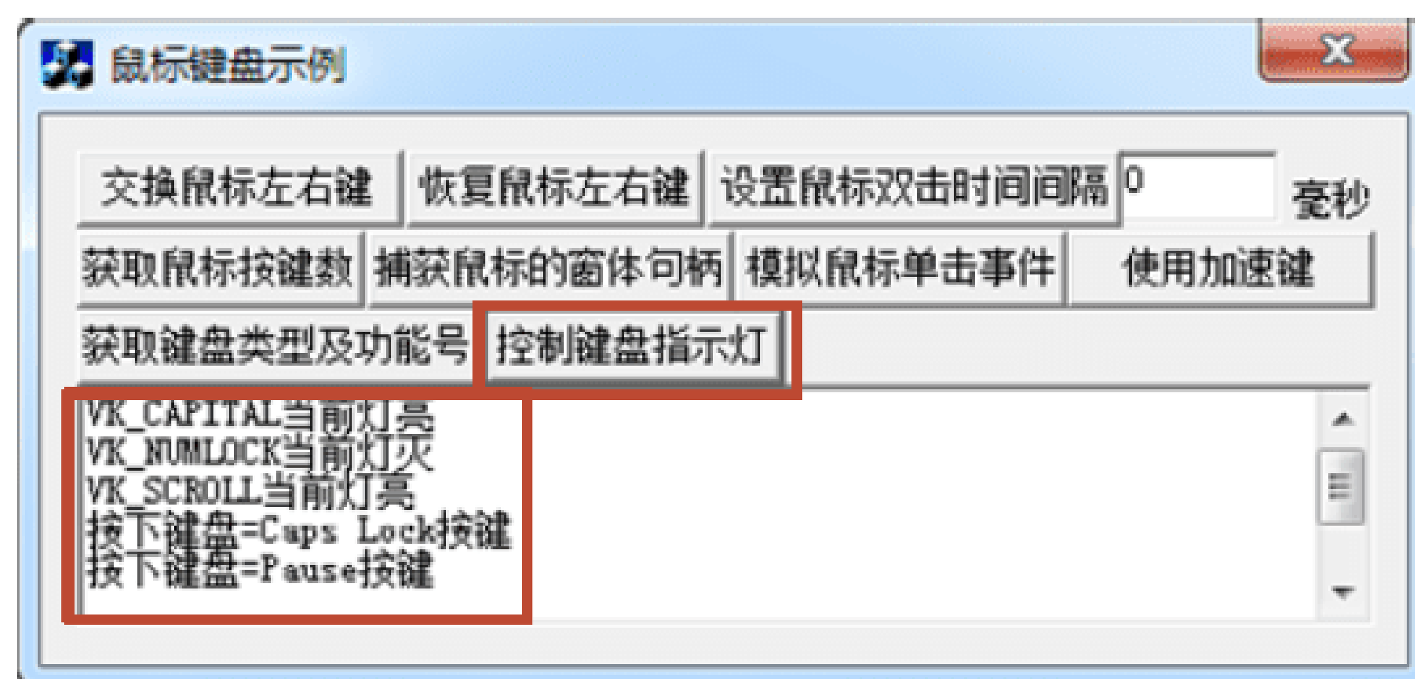


图 20-80　控制键盘指示灯的运行效果

20.11　本 章 小 结

本章介绍了与系统相关的编程，主要包括磁盘信息操作的方法、系统控制与调用的方法、应用程序操作、系统工具的调用方法、桌面相关信息和系统信息的操作方法、消息与剪贴板的使用以及鼠标键盘的设置。本章的重点是掌握在实际项目中实现常用系统功能的方法。本章的难点是掌握调用 Windows 底层接口的方法。第 21 章将介绍注册表、INI 和 XML 文件操作。

20.12　习　　题

1．本章 20.1.7 小节讲解了如何使用 GetDiskFreeSpace()函数来获取磁盘控件的信息，那么如何使用此函数的升级版 GetDiskFreeSpaceEx()呢？

【思路】参看 MSDN 中对函数 GetDiskFreeSpaceEx()的使用介绍。

2．本章 20.2.9 小节讲解了如何监视指定目录中的文件名称改变的操作。尝试编程来监视指定目录中的文件写操作。

【思路】参考 20.2.9 小节的示例。

3．本章 20.3.1 小节讲解了如何调用记事本程序。尝试模仿该示例，调用 Windows 的画图程序（画图程序的路径：C:\Windows\system32\mspaint.exe）。

【思路】参考 20.3.1 小节的示例。

4．本章 20.4.1 小节讲解了通过互斥对象来禁止程序重复运行的方法。尝试使用事件对象来实现同样的功能。

【思路】创建事件对象的 API 是 CreateEvent()函数，在 MSDN 中查找此函数的使用方式，进而完成要求的功能。

第 21 章　注册表、INI 和 XML 文件

在 Windows 操作系统中，存储系统信息和应用程序信息有两种常用方式，通过注册表和使用 INI 文件。通常情况下，系统会将组件信息和系统信息存储在注册表中，而应用程序的配置信息会使用 INI 文件存储。随着 XML 标准的发展，XML 文件也成为一种常见的存储配置信息的格式。本章将介绍有关这 3 方面的知识。

21.1　读写注册表的 API 函数

注册表是系统定义的用于存储应用程序和系统组件配置信息的数据库。Windows 操作系统中的许多信息都是存储在注册表中的，包括组件信息和系统信息。本节将介绍在 Visual Studio 2010 环境下操作注册表的 API 函数。

21.1.1　注册表的概念

注册表以树结构存储数据，树中的每个结点称为键，每个键可以包含子键（值域）和值对，而键值可以是任何类型的值。每个子键描述了键的某个属性的取值。通常情况下，每个应用程序会打开一个键并使用与此键相关的值。键名是由一个或多个字符组成，以点号（.）开头的键名是系统预留的键名。

虽然在注册表中存储应用程序数据没有什么类型和大小的限制，但是为了提高系统和程序性能，应该遵守一些约定俗成的习惯。一般应用程序会将配置信息和初始化数据存储在注册表中，而其他类型的数据则以文件方式存储。如果数据超过 1KB 或 2KB，则应该将其存储在文件中，并使用注册表中的键指向文件，而不是将数据存储为注册表键值。如果数据量比较大，则应该将数据存储为文件，并直接引用文件。尤其是可执行的二进制代码不能存储在注册表中。

注册表条目值占用的空间比注册表键占用的空间少。为了节约空间，应用程序应该将相同的数据分组到结构中，并将结构作为值存储，而不是将结构的每个成员作为单独键存储。

应用程序在向注册表中添加数据时，必须首先打开注册表键。要打开一个键，应用程序必须提供已经存在的注册表键的句柄。系统定义了几个总是打开的标准的注册表键。应用程序可以使用这些预定义的句柄作为注册表的入口。系统提供了两个注册表的预定义根键，即 HKEY_LOCAL_MACHINE 和 HKEY_USERS。另外，还定义了 HKEY_CLASSES_ROOT（HKEY_LOCAL_MACHINE 的子键）和 HKEY_CURRENT_USER（HKEY_USERS 的子

键）。这些注册表句柄在所有的 Win32 注册表实现中都是可用的，但是各个平台之间的处理是不同的。预定义键帮助程序在注册表中导航，并允许系统管理员操作不同种类的数据。如表 21-1 所示中列出了注册表入口的预定义键。

表 21-1　预定义注册表键

入　　口	用　　途
HKEY_CLASSES_ROOT	此注册表键下的条目用于定义文档类型以及与这些类型相关的属性。由 shell 程序和 OLE 应用程序使用
HKEY_CURRENT_USER	此注册表键下的条目用于定义当前用户的参数设置。这些参数设置包括环境变量设置、程序分组数据、颜色、打印机、网络连接和应用程序等参数
HKEY_LOCAL_MACHINE	此注册表键下的条目用于定义计算机的物理状态，包括总线类型数据、系统内存以及安装的硬件和软件
HKEY_USERS	此注册表键下的条目用于定义本地计算机上的新用户的默认用户配置和当前用户配置

在将数据存储到注册表前，程序首先应该将数据分为两种：与计算机有关的数据和与用户有关的数据。程序通过区分这些数据可以支持多用户的不同配置。将与计算机有关的数据记录在 HKEY_LOCAL_MACHINE 键下，如可以创建键存储公司名称、产品名称和版本号，代码如下：

```
HKEY_LOCAL_MACHINE\Software\TestCompany\TestProduct \2.0
```

如果应用程序支持 OLE，则应该将数据记录在：

```
HKEY_LOCAL_MACHINE\Software\Classes.
```

与用户相关的数据应该记录在 HKEY_CURRENT_USER 下，代码如下：

```
HKEY_CURRENT_USER\Software\ TestCompany\TestProduct \2.0
```

21.1.2　创建带安全属性的注册表项

在 Win32 中，使用 RegCreateKeyEx()函数可以创建带有安全属性的注册表项。如果此注册表项已经存在，则函数会打开此注册表项。其函数原型为：

```
LONG RegCreateKeyEx(
  HKEY hKey,                    //当前打开的注册表项或预定义的注册表项句柄值
  LPCTSTR lpSubKey,             //指定此函数打开或创建的子键的名称
  DWORD Reserved,               //预留
  LPTSTR lpClass,               //指定键的类名
  DWORD dwOptions,              //指定键的选项
  REGSAM samDesired,            //指定新键的访问安全选项
  LPSECURITY ATTRIBUTES lpSecurityAttributes,
                                //确定返回的句柄是否可以被子进程继承
  PHKEY phkResult,              //存储打开或创建的键的句柄
  LPDWORD lpdwDisposition);     //存储函数执行的操作的方式，是创建新键还是打开键
```

其中，dwOptions 参数指定此键的创建选项，有效值如表 21-2 所示。

表 21-2　创建注册表键的选项

值	含　义
REG_OPTION_NON_VOLATILE	默认值，表示创建的注册表键是不易丢失的。信息存储在文件中，并且当系统重启后可以继续使用。RegSaveKey()函数可以保存使用此选项创建的注册表项
REG_OPTION_VOLATILE	创建的注册表键是临时的，存储在内存中，当系统重启后就不存在了。RegSaveKey()函数不能保存此选项指定的注册表项。当指定的注册表项已经存在时，会忽略此选项
REG_OPTION_BACKUP_RESTORE	此选项表示函数会忽略 samDesired 参数，并且使用备份和恢复权限打开注册表

samDesired 参数指定新键的访问安全选项，取值可以是如表 21-3 所示的各个选项的组合

表 21-3　注册表键的安全选项

值	含　义
KEY_ALL_ACCESS	KEY_QUERY_VALUE 、 KEY_ENUMERATE_SUB_KEYS 、 KEY_NOTIFY、KEY_CREATE_SUB_KEY、KEY_CREATE_LINK 和 KEY_SET_VALUE 权限的组合
KEY_CREATE_LINK	允许创建符号链接
KEY_CREATE_SUB_KEY	允许创建子键
KEY_ENUMERATE_SUB_KEYS	允许枚举子键
KEY_EXECUTE	允许读取注册表键
KEY_NOTIFY	允许发送修改通知
KEY_QUERY_VALUE	允许查询子键数据
KEY_READ	KEY_QUERY_VALUE 、 KEY_ENUMERATE_SUB_KEYS 和 KEY_NOTIFY 权限的组合
KEY_SET_VALUE	允许设置子键数据
KEY_WRITE	KEY_SET_VALUE 和 KEY_CREATE_SUB_KEY 权限的组合

调用此函数创建的注册表键没有取值，应用程序应该调用 RegSetValue()函数或 RegSetValueEx()函数设置键值。需要注意的是，不能在 HKEY_USERS 键或 HKEY_LOCAL_MACHINE 键下创建新建。

21.1.3　创建注册表项

调用 RegCreateKey()函数可以快速地创建指定的键。如果在注册表中已经存在键，则函数会打开此键。其函数原型为：

```
LONG RegCreateKey(
  HKEY hKey,                //当前打开的键的句柄或预定义的注册表句柄值
  LPCTSTR lpSubKey,         //要打开或创建的注册表键相对于 hKey 的子键的名称
  PHKEY phkResult );        //指向打开或创建的键的句柄
```

调用此函数可以创建多级键值，如 subkey1\subkey2\subkey3\subkey4。hKey 参数指定的键在打开时，必须使用 KEY_CREATE_SUB_KEY 权限打开，才可以使用此函数创建新

注册表键。下面的代码显示了使用此函数快速创建注册表项的方法。

```
01  void CreateKeyTest()                                  //快速创建注册表键
02  {
03      HKEY hKey;                                        //定义注册表键变量
04      //创建新注册表键
05      int iRet = RegCreateKey(HKEY_LOCAL_MACHINE,
06          "SOFTWARE\\LLN\\VC\\Registry\\CreateTest", &hKey);
07      if (iRet == ERROR_SUCCESS)
08          printf("快速创建注册表成功");
09      else
10          printf("快速创建注册表失败，错误代码=%d", iRet);
11  }
```

上面代码使用 RegCreateKey()函数在 HKEY_LOCAL_MACHINE 注册表项下创建了新的注册表项 SOFTWARE\LLN\VC\Registry\CreateTest。程序运行后，注册表如图 21-1 所示。



图 21-1　创建注册表项后的效果

21.1.4　打开注册表项

21.1.3 小节介绍过调用 RegCreateKey()函数或 RegCreateKeyEx()函数可以创建注册表键，而调用 RegOpenKey()函数或 RegOpenKeyEx()函数可以打开注册表键。使用完注册表键后，应该调用 RegCloseKey()函数关闭注册表键，此函数在返回之前，并没有将数据写回注册表键，可以使用 RegFlushKey()函数显式地将数据写回注册表，但是此函数占用系统资源比较多，所以除非必须，否则不建议使用此函数。调用 RegOpenKey()函数可以打开指定的注册表键。其函数原型为：

```
LONG RegOpenKey(
  HKEY hKey,                  //表示当前打开的键的句柄或是预定义的注册表句柄值
  LPCTSTR lpSubKey,           //表示此函数要打开的注册表键相对于 hKey 的子键的名称
  PHKEY phkResult);           //指向打开的注册表键的句柄
```

下面的代码显示了使用此函数快速打开注册表项的方法。

```
01  void OpenKeyTest()                    //打开注册表键
02  {
03      HKEY hKey;                        //定义注册表键变量
04      //打开注册表键
05      int iResult = RegOpenKey(HKEY_LOCAL_MACHINE,
06          "SOFTWARE\\LLN\\VC\\Registry\\CreateTest", &hKey);
07      if (iResult == ERROR_SUCCESS)
```

```
08          //输出成功信息
09          printf("打开注册表成功.HKEY=%d", hKey);
10      else
11          printf("打开注册表失败，错误代码=%d", iResult);
12  }
```

上面的代码使用 RegOpenKey()函数打开 HKEY_LOCAL_MACHINE 下的注册表项 SOFTWARE\LLN\VC\Registry\CreateTest。程序运行后，效果如图 21-2 所示。

21.1.5　判断注册表项是否存在

通过 RegOpenKey()函数的返回值，可以判断注册表项是否存在。如果返回值为 ERROR_FILE_NOT_FOUND，则表示要打开的注册表不存在。代码如下：

```
01  bool IfKeyExistTest()                              //判断注册表项是否存在
02  {
03      HKEY hKey;                                     //注册表键句柄
04      //打开注册表键
05      long lResult =RegOpenKey(HKEY_LOCAL_MACHINE,
06          "SOFTWARE\\LLN\\VC\\Registry\\CreateTest", &hKey);
07      if (lResult == ERROR_FILE_NOT_FOUND)
08      {
09          printf("指定的注册表项不存在");
10          return false;
11      }
12      else
13      {
14          printf("指定的注册表项存在");
15          return true;
16      }
17  }
```

上面代码判断注册表中是否存在 SOFTWARE\LLN\VC\Registry\CreateTest 注册表项。程序运行效果如图 21-3 所示。

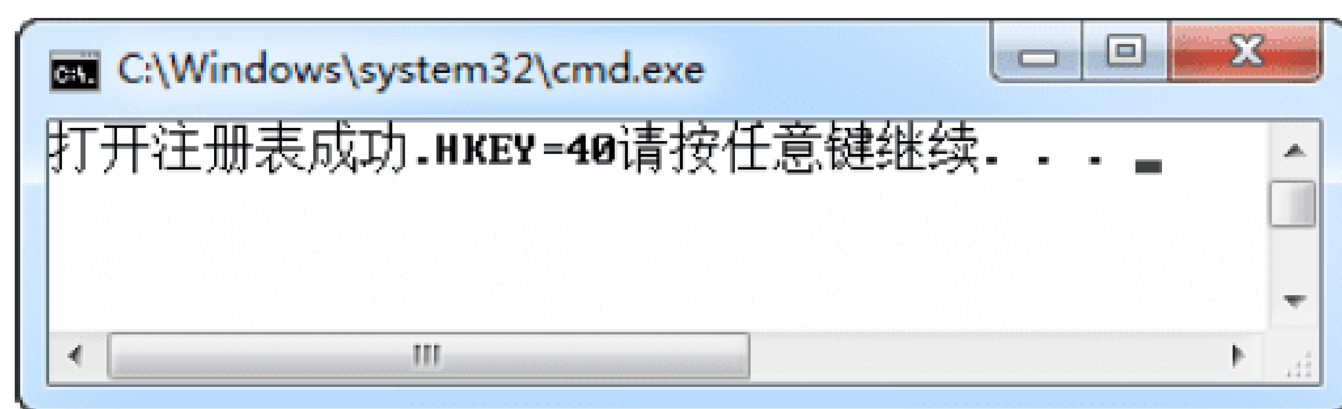


图 21-2　打开注册表项后的效果

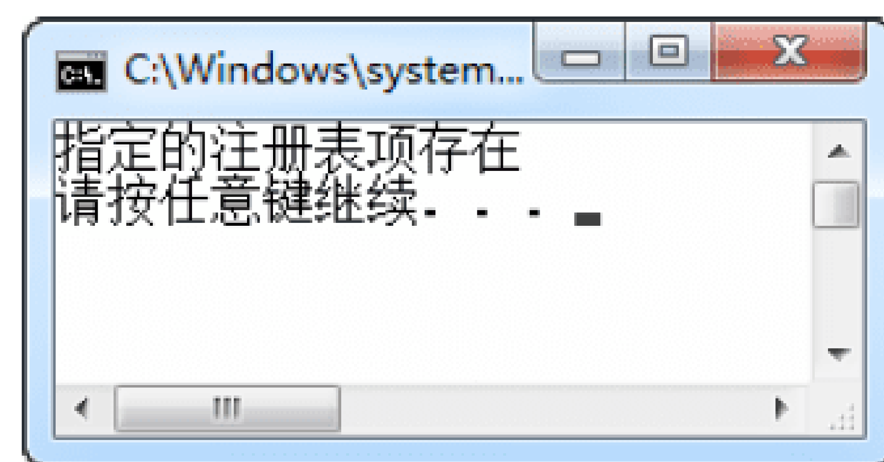


图 21-3　判断注册表项是否存在运行效果

21.1.6　删除注册表项

使用 RegDeleteKey()函数可以删除注册表项。其函数原型为：

```
LONG RegDeleteKey(
  HKEY hKey,
  LPCTSTR lpSubKey );
```

其中 hKey 参数表示当前打开的键的句柄或是预定义的注册表句柄值，调用此函数删除的键是此键的子键。lpSubKey 参数表示此函数要删除的注册表键相对于 hKey 的子键的

名称。如果函数操作成功，则返回 ERROR_SUCCESS；否则返回非 0。下面是删除注册表项的示例。

```
01  bool DeleteKeyTest()                          //删除注册表项
02  {
03      HKEY myKey;                               //要删除的注册表项句柄
04      //删除
05      if(RegDeleteKey(HKEY_LOCAL_MACHINE,
06          "SOFTWARE\\LLN\\VC\\Registry\\CreateTest") == ERROR_SUCCESS)
07      {
08          printf("删除注册表项成功");
09          return true;
10      }
11      else
12      {
13          printf("删除注册表项失败");
14          return false;
15      }
16  }
```

上面代码调用 RegDeleteKey() 函数删除了 HKEY_LOCAL_MACHINE 下的 SOFTWARE\LLN\VC\Registry\CreateTest 注册表键。程序运行效果如图 21-4 所示。

21.1.7　打开注册表根项

打开注册表根项的方法与打开注册表项的方法相同。不同之处在于，要打开注册表根项，需要将 RegOpenKey()函数的第二个参数设置为 NULL，则函数会打开与第一个参数相同的注册表根项。代码如下：

```
01  bool OpenRootKeyTest()                                //打开注册表根项
02  {
03      HKEY hKey;                                        //注册表键值
04      //打开根项
05      int iResult = RegOpenKey(HKEY_LOCAL_MACHINE, NULL, &hKey);
06      if (iResult == ERROR_SUCCESS)
07      {
08          printf("打开注册表根项成功");
09          return true;
10      }
11      else
12      {
13          printf("打开注册表根项失败");
14          return false;
15      }
16  }
```

上面代码会打开 HKEY_LOCAL_MACHINE 注册表根项。程序运行效果如图 21-5 所示。

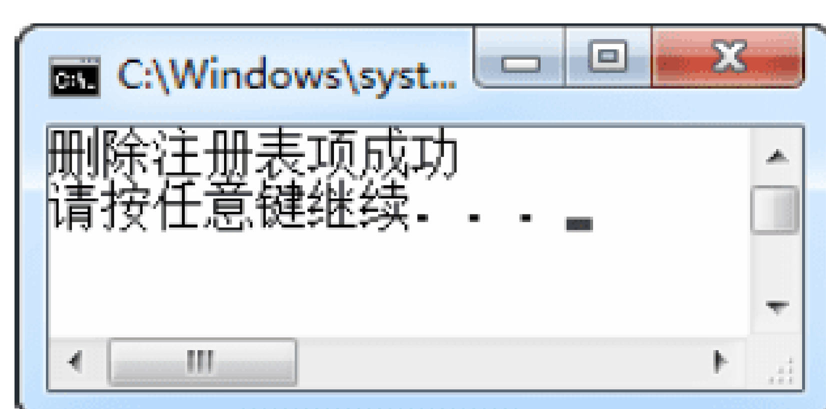


图 21-4　删除注册表项运行效果

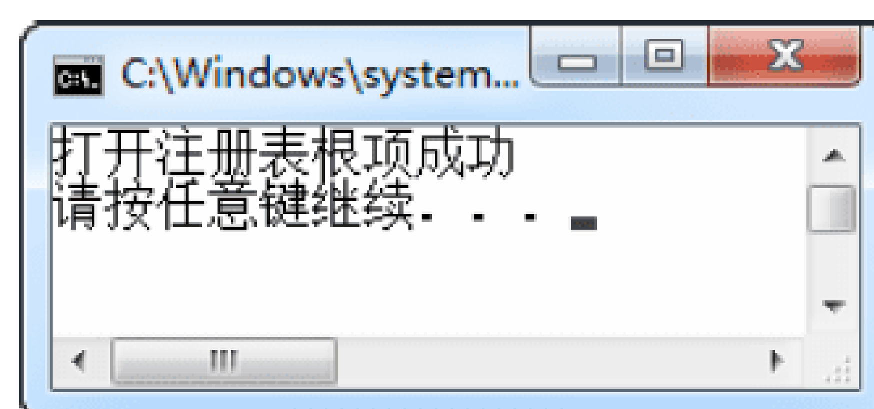


图 21-5　打开注册表根项运行效果

21.1.8　指定注册表项的默认值

调用 RegSetValueEx()函数可以向指定注册表项写入值。在使用此函数前，首先需要打开注册表项。其函数原型为：

```
LONG RegSetValueEx(
  HKEY hKey,                       //打开的注册表项的句柄
  LPCTSTR lpValueName,             //要写入值的注册表项的键值
  DWORD Reserved,                  //预留参数
  DWORD dwType,                    //要写入的数据的类型
  CONST BYTE *lpData,              //指向要写入的数据的指针
  DWORD cbData);                   //表示要写入的数据的长度
```

其中，dwType 参数表示要写入的数据的类型，有效取值如表 21-4 所示。

表 21-4　注册表键值的取值的数据类型

取　　值	数 据 类 型
REG_BINARY	二进制数据
REG_DWORD	32 位整数
REG_DWORD_LITTLE_ENDIAN	低位在前的整数
REG_DWORD_BIG_ENDIAN	高位在前的整数
REG_EXPAND_SZ	字符串
REG_LINK	Unicode 符号链接
REG_MULTI_SZ	字符串数组，以两个 NULL 结束
REG_NONE	未定义的值类型
REG_RESOURCE_LIST	设备驱动资源列表
REG_SZ	字符串

调用完该函数后，应该调用 RegCloseKey()函数将此句柄关闭。如果函数操作成功，则返回 ERROR_SUCCESS；否则返回非 0。代码如下：

```
01  bool WriteKeyDefaultValueTest()                        //向关键值写入默认值
02  {
03      DWORD value = 21;                                  //定义写入值变量
04      HKEY hKey;                                         //定义键值句柄
05      if (RegOpenKey(HKEY_LOCAL_MACHINE,
06          "SOFTWARE\\LLN\\VC\\Registry\\CreateTest",
07          &hKey) != ERROR_SUCCESS)                       //打开注册表项
08      {
09          printf("打开注册表项失败");
10          return false;
11      }
12      bool bRet;                                         //定义返回值
13      if(RegSetValueEx(hKey, NULL, NULL, REG_DWORD,
14          (const BYTE*)&value,sizeof(DWORD))
15          == ERROR_SUCCESS)                   //写入默认值
16      {
17          printf("向指定注册表项写入默认值成功");
18          bRet = true;
19      }
```

```
20      else
21      {
22          printf("向指定注册表项写入默认值失败");
23          bRet = false;
24      }
25      RegCloseKey(hKey);                              //关闭注册表项
26      return bRet;                                    //函数返回
27  }
```

上面代码首先打开HKEY_LOCAL_MACHINE注册表中的SOFTWARE\LLN\VC\Registry\CreateTest 项，将数值 21 写入此键值并返回。程序运行效果如图 21-6 所示。

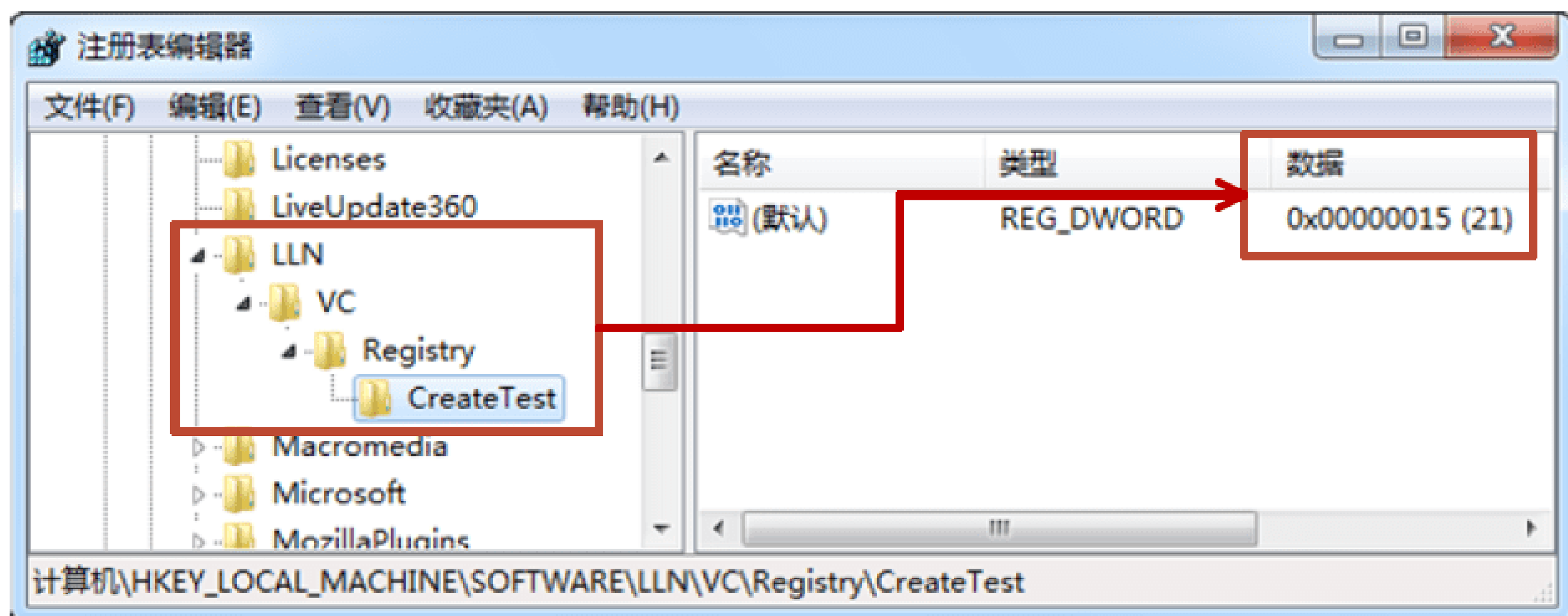


图 21-6　向指定注册表项写入默认键值运行效果

21.1.9　设置注册表键值

设置注册表键值数据与 21.1.8 小节介绍的向指定注册表项默认键值写入数据的方法是一样的。有两点需要注意，一是需要根据数据类型的不同，指定 RegSetValueEx()函数的第四个参数为相应的数据类型；另外需要在第二个参数中指定键值的名称。此处指定键值名称为 MaxVersion，代码如下：

```
01  bool WriteKeyValueTest()                            //设置注册表键值
02  {
03      DWORD value = 5;                                //定义写入的键值变量
04      HKEY hKey;                                      //定义注册表项句柄
05      if (RegOpenKey(HKEY_LOCAL_MACHINE,
06          "SOFTWARE\\LLN\\VC\\Registry\\CreateTest",
07          &hKey) != ERROR_SUCCESS)                    //打开注册表项
08      {
09          printf("打开注册表项失败");
10          return false;
11      }
12      bool bRet;                                      //定义返回值
13      if(RegSetValueEx(hKey, "MaxVersion", NULL, REG_DWORD,
14          (const BYTE*)&value, sizeof(DWORD)) == ERROR_SUCCESS)
15      {
16          printf("向指定注册表项写入键值成功");
17          bRet = true;
18      }
19      else
```

```
20       {
21           printf("向指定注册表项写入键值失败");
22           bRet = false;
23       }
24       RegCloseKey(hKey);                          //关闭注册表键值
25       return bRet;                                //函数返回
26   }
```

上面程序为 SOFTWARE\LLN\VC\Registry\CreateTest 注册表项的 MaxVersion 键值写入整型值 5。程序运行效果如图 21-7 所示。

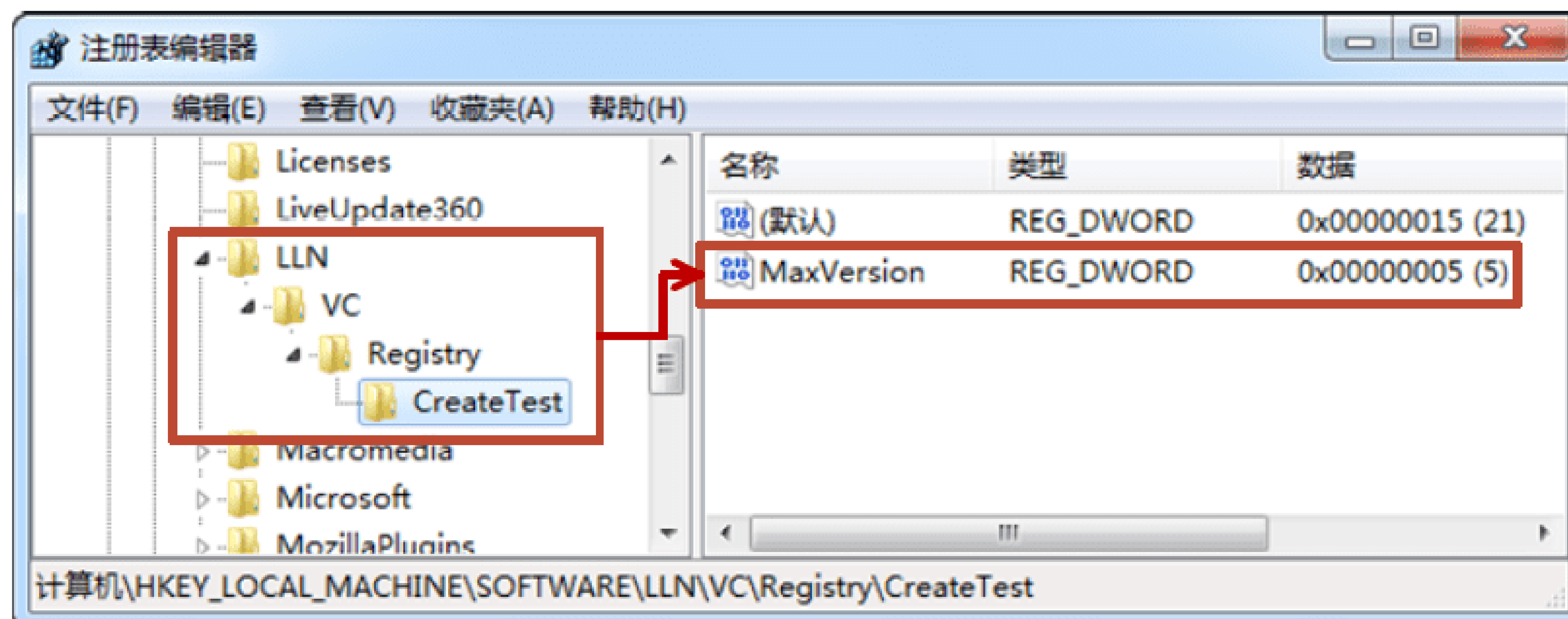


图 21-7　设置注册表键值数据运行效果

21.1.10　快速设置注册表键值字符串

21.1.9 小节介绍了向注册表键值写入数据的方法，但是写入数据前，需要首先打开注册表项。除了使用这种方式，还可以调用 RegSetValue()函数直接设置注册表键值。其函数原型为：

```
LONG RegSetValue(
  HKEY hKey,               //写入键值数据的打开的注册表项或预定义的注册表项句柄值
  LPCTSTR lpSubKey,        //指定此函数要写入数据的键值的名称
  DWORD dwType,            //要写入的数据的类型
  LPCTSTR lpData,          //指向要写入的数据的指针
  DWORD cbData);           //表示要写入的数据的长度
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则返回非 0。代码如下：

```
01   bool QuickWriteKeyValueTest()          //快速设置注册表键值示例
02   {
03       char author[50]={0};               //写入的数值变量
04       strcpy(author, "杯子");            //为写入变量赋值
05       bool bRet = false;                 //定义返回值
06       if (RegSetValue(HKEY_LOCAL_MACHINE,
07           "SOFTWARE\\LLN\\VC\\Registry\\CreateTest",
08           REG_SZ, (const char*)author, strlen(author))
09           == ERROR_SUCCESS)
10       {
11           printf("快速设置注册表键值字符串数据成功");
12           bRet = true;
13       }
14       else
15       {
16           printf("快速设置注册表键值字符串数据失败");
17           bRet = false;
```

```
18      }
19      return bRet;
20  }
```

上面程序为 SOFTWARE\LLN\VC\Registry\CreateTest 注册表项的默认键值写入字符串值“杯子”。程序运行效果如图 21-8 所示。

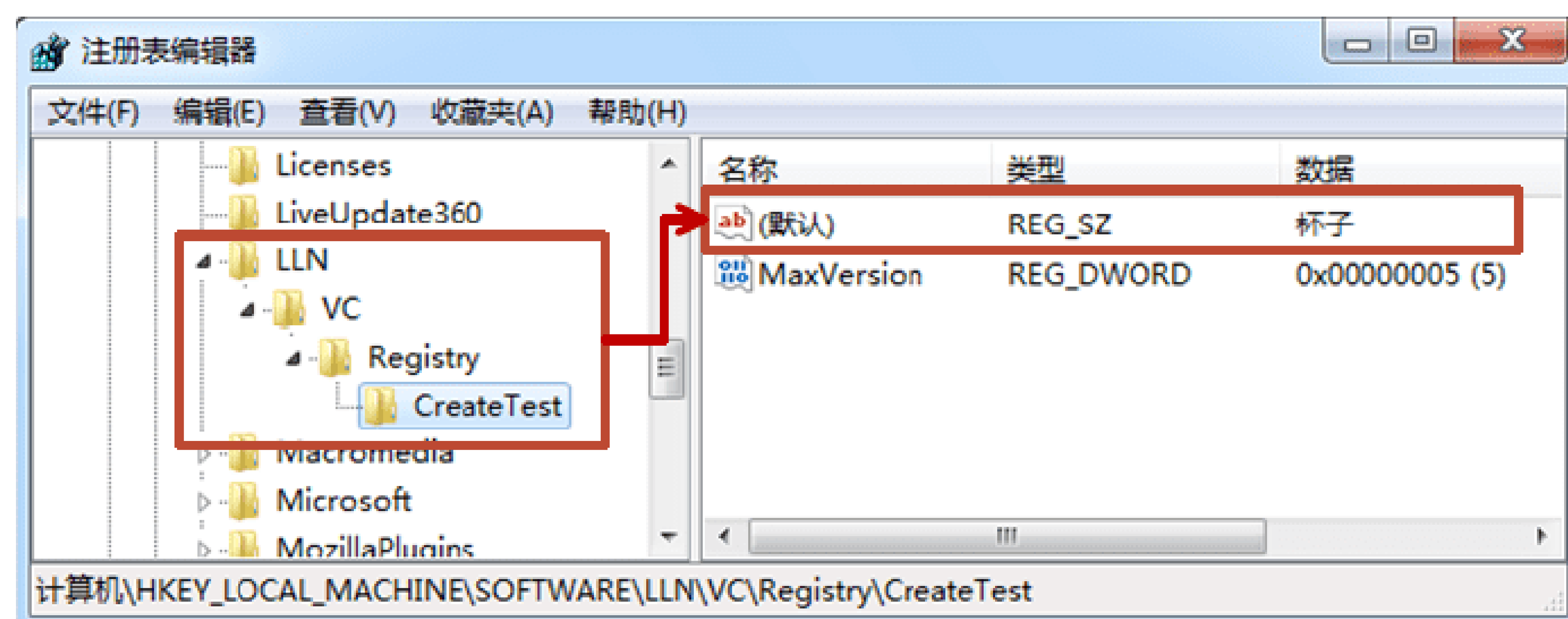


图 21-8　快速设置注册表键值字符串数据运行效果

21.2　注册表应用

因为 Windows 操作系统将部分信息记录在注册表中，因此通过操作注册表可以实现一些系统功能，即有关注册表的应用。这些应用包括隐藏/显示回收站、隐藏/显示我的电脑、隐藏/显示驱动器、禁用部分菜单、禁用文件、保存注册表项、设置开机自动运行程序以及清除历史文档记录等。本节中将介绍这些应用的实现。

21.2.1　保存注册表项

调用 RegSaveKey()函数可以将注册表项的内容保存到文件中。但是在保存注册表前，首先需要提升线程的操作权限，为其增加 SE_BACKUP_NAME 权限，否则会出现无法成功保存注册表项的情况。RegSaveKey()函数原型如下：

```
LONG RegSaveKey(
  HKEY hKey,                        //要保存的注册表项的句柄
  LPCTSTR lpFile,                   //要将注册表项保存到文件的文件名
  LPSECURITY_ATTRIBUTES lpSecurityAttributes); //有关安全属性的选项设置
```

下面是实现保存注册表项的代码。

```
01  void SaveRegKey()                          //保存注册表项
02  {
03      TOKEN_PRIVILEGES tp;                   //定义权限变量
04      HANDLE hToken;                         //权限句柄
05      LUID luid;                             //定义 LUID
06      if(!OpenProcessToken(GetCurrentProcess(),
07          TOKEN_ADJUST_PRIVILEGES,&hToken ))
08      {
09          //打开进程权限句柄
10          printf("调用 OpenProcessToken 失败");
11          return;
```

```
12        }
13        //查找 SE_BACKUP_NAME 权限值
14        if(!LookupPrivilegeValue(NULL, SE_BACKUP_NAME, &luid))
15        {
16            printf("调用 LookupPrivilegeValue 失败");
17            return;
18        }
19        tp.PrivilegeCount          = 1;      //定义权限调整数量
20        tp.Privileges[0].Luid       = luid;  //设置权限值
21        //设置打开此权限
22        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
23        //调整进程权限
24        AdjustTokenPrivileges(hToken, false, &tp,
25            sizeof(TOKEN_PRIVILEGES),NULL, NULL );
26        if (GetLastError() != ERROR_SUCCESS)//如果调整进程失败，则返回
27        {
28            printf("调用 AdjustTokenPrivileges 失败");
29            return;
30        }
31        HKEY hKeyToSave;                     //定义要保存的键句柄
32        DWORD dwDisposition;                 //定义选项值
33        //创建并打开注册表项
34        if(RegCreateKeyEx(HKEY_LOCAL_MACHINE,
35            "SOFTWARE\\LLN\\VC\\Registry\\CreateTest", 0,NULL,
36            REG_OPTION_BACKUP_RESTORE,KEY_ALL_ACCESS, NULL,
37            &hKeyToSave,&dwDisposition)==ERROR_SUCCESS)
38        {
39            //保存注册表项到文件中
40            if (RegSaveKey(hKeyToSave,
41                   "C:\\MyInfo.reg", NULL) == ERROR_SUCCESS)
42                printf("保存注册表成功");
43            else
44                printf("保存注册表失败");
45            RegCloseKey(hKeyToSave);
46        }
47        else
48        {
49            printf("打开注册表失败");
50        }
51        return;       //函数返回
52    }
```

上面的程序将 SOFTWARE\LLN\VC\Registry\CreateTest 注册表中的内容保存到文件 MyInfo.reg 文件中。程序运行效果如图 21-9 所示。

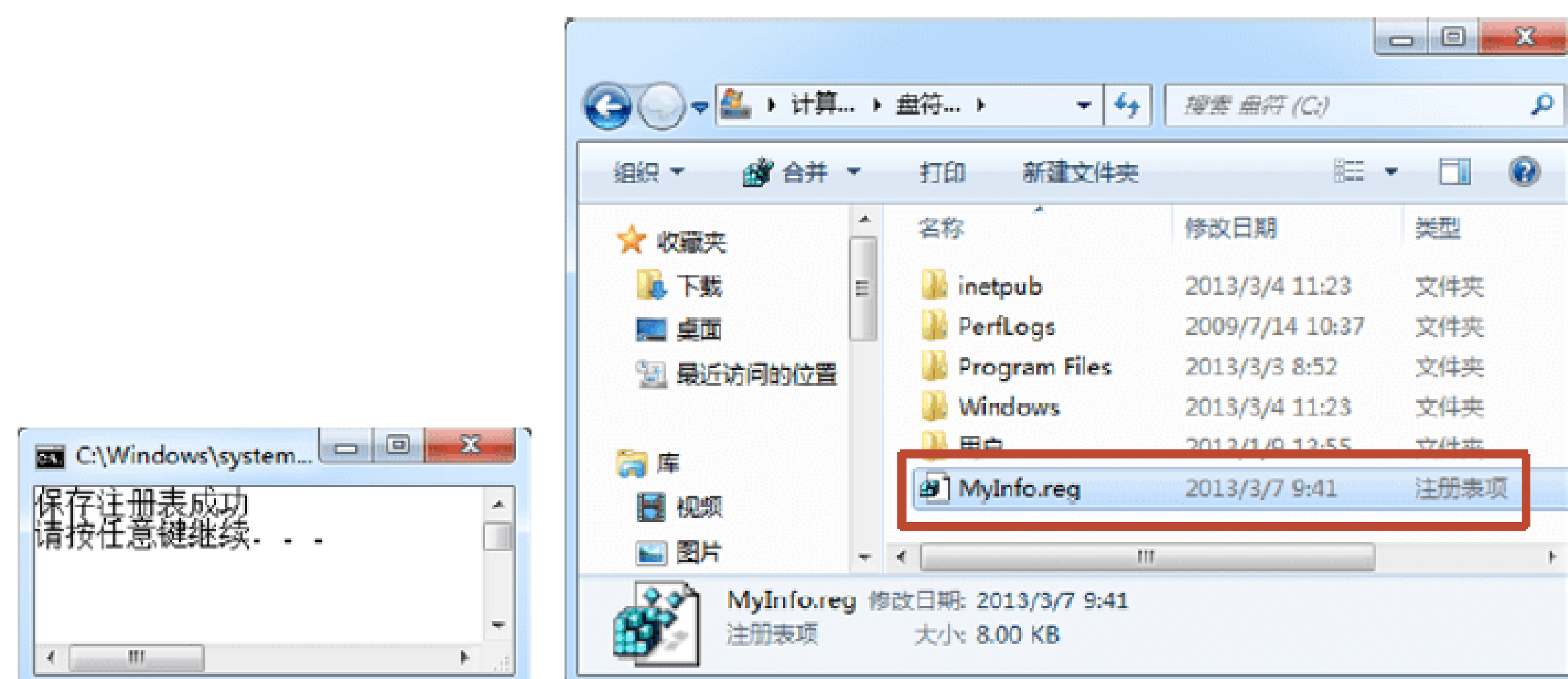


图 21-9　保存注册表项运行效果

21.2.2　开机自动运行

要使得程序在开机自动运行，可以通过修改注册表实现。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
```

在上面的注册表项下建立对应的键值对就可以，键名为程序名，键对应的值名称为要运行的程序的完整文件名。代码如下：

```
01  void SysStartRun()                          //开机自动运行
02  {
03      char szValue[MAX_PATH]={0};             //定义开机运行文件名变量
04      strcpy(szValue, "C:\\Hello.exe");       //为开机运行文件赋值
05      HKEY hKey;                              //注册表项句柄
06      //打开注册表项
07      if (RegOpenKey(HKEY_CURRENT_USER,
08          "Software\\Microsoft\\Windows\\CurrentVersion\\Run",
09          &hKey) == ERROR_SUCCESS)
10      {
11          //添加启动项
12          if(RegSetValueEx(hKey, "Hello", NULL, REG_SZ,
13              (const BYTE*)&szValue,strlen(szValue))
14              == ERROR_SUCCESS)
15              printf("添加开机自动运行项成功");
16          else
17              printf("添加开机自动运行项失败");
18          RegCloseKey(hKey);                  //关闭注册表项
19      }
20      else
21      {
22          printf("打开注册表项失败");
23      }
24  }
```

上面的代码使得C盘下的Hello.exe程序在开机时自动运行。程序运行效果如图21-10所示。

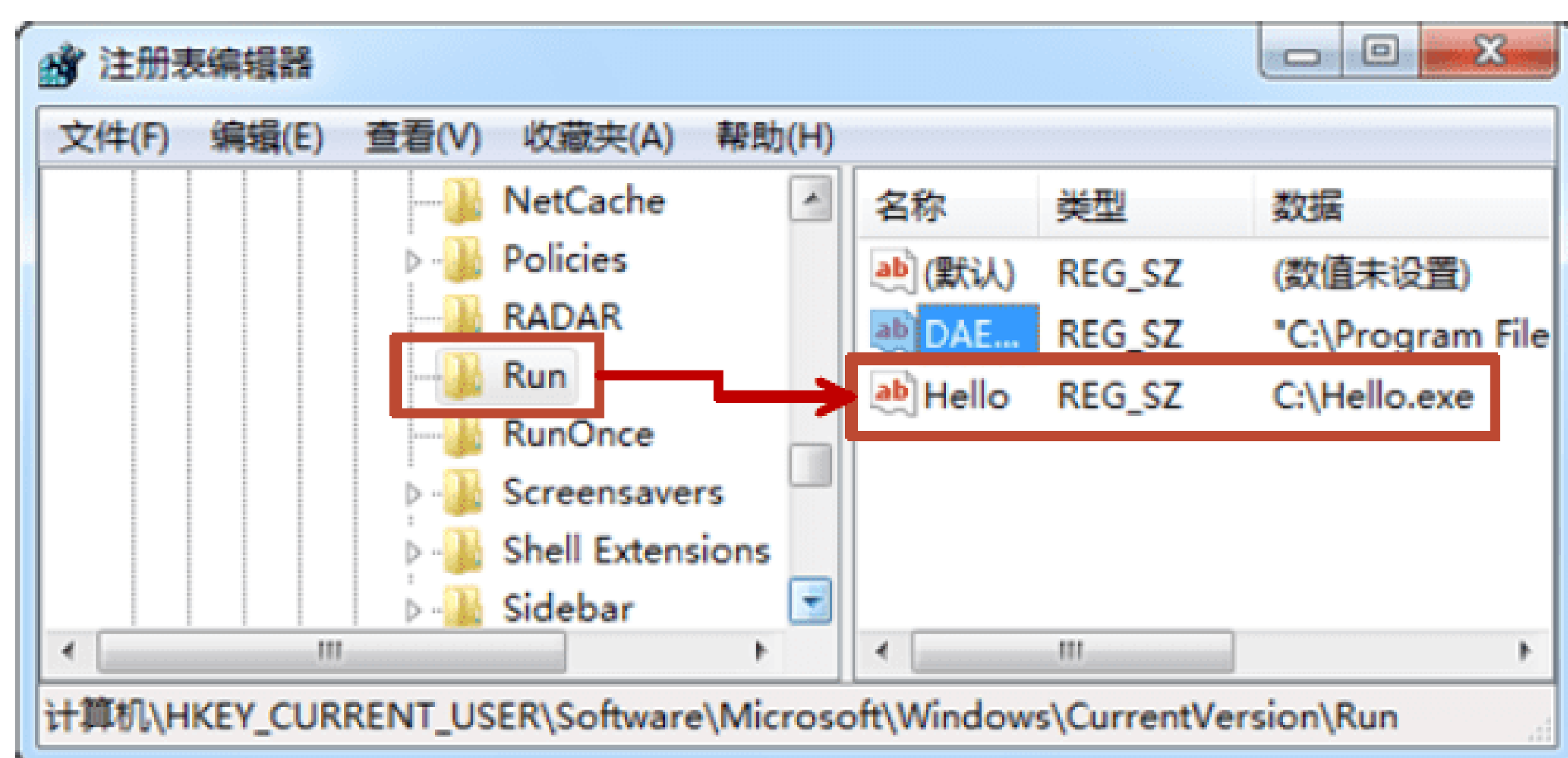


图 21-10　设置开机自动运行

21.2.3　隐藏和显示我的电脑

要隐藏和显示我的电脑，可以通过修改注册表实现。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\No
nEnum
```

在上面的注册表项中，建立名称为{20D04FE0-3AEA-1069-A2D8-08002B30309D}的键。当其值设置为 1 时，会隐藏我的电脑；当其值设置为 0 时，会显示我的电脑。代码如下：

```
01  void HideMyComputer(BOOL bHide)          //隐藏和显示我的电脑
02  {
03      DWORD dwValue;                        //定义设置的取值
04      if (bHide) dwValue = 1;               //如果隐藏
05      else dwValue = 0;                     //如果显示
06      HKEY hKey;                            //注册表键句柄
07      if (RegCreateKey(HKEY_CURRENT_USER,
08          "Software\\Microsoft\\Windows\\
09           CurrentVersion\\Policies\\NonEnum",
10          &hKey) == ERROR_SUCCESS)          //创建注册表项
11      {
12          //设置注册表值
13          if(RegSetValueEx(hKey,
14              "{20D04FE0-3AEA-1069-A2D8-08002B30309D}",
15              NULL,REG_DWORD, (const BYTE*)&dwValue, sizeof(DWORD))
16              == ERROR_SUCCESS)
17              printf("%s 我的电脑成功", dwValue==1?"隐藏":"显示");
18          else
19              printf("%s 我的电脑失败", dwValue==1?"隐藏":"显示");
20          RegCloseKey(hKey);                //关闭注册表项
21      }
22      else
23          printf("创建注册表值失败");
24  }
```

运行上面的代码，则会根据传入的参数显示或隐藏我的电脑。程序运行效果如图 21-11 所示，同时会隐藏我的电脑。

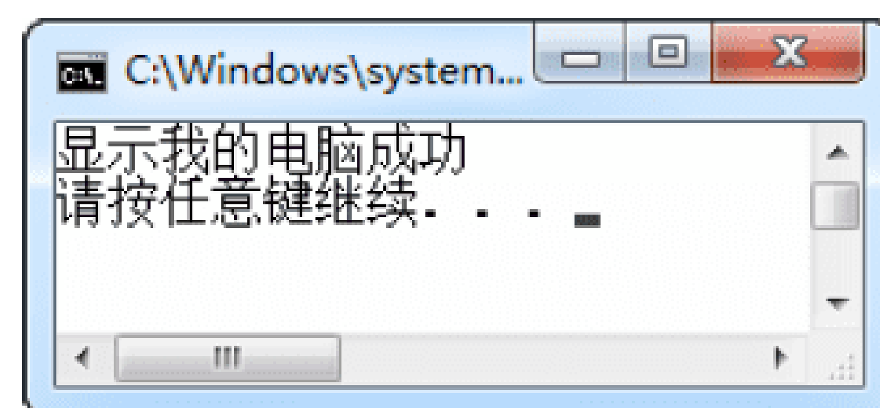


图 21-11　隐藏和显示我的电脑运行效果

21.2.4　隐藏和显示回收站

要隐藏和显示回收站，可以通过修改注册表实现。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\hi
dedesktopicons\newstartpanel
```

在上面的注册表项中，建立名称为{645ff040-5081-101b-9f08-00aa002f954e}的键。当其值设置为 1 时，会隐藏回收站；当其值设置为 0 时，会显示回收站。代码如下：

```
01  void HideRcyBin(BOOL bHide)              //隐藏和显示回收站
02  {
03      DWORD value;                          //定义设置的取值
04      if (bHide)
05          value = 1;                        //如果隐藏
06      else
07          value = 0;                        //如果显示
08      HKEY hKey;                            //注册表键句柄
09      if (RegCreateKey(HKEY_CURRENT_USER,
10          "Software\\microsoft\\windows\\currentversion\\
```

```
11          explorer\\hidedesktopicons\\newstartpanel",
12          &hKey) == ERROR_SUCCESS)         //创建注册表项
13      {
14          //设置注册表值
15          if(RegSetValueEx(hKey,
16              "{645ff040-5081-101b-9f08-00aa002f954e}",
17              NULL, REG_DWORD, (const BYTE*)&value, sizeof(DWORD))
18              == ERROR_SUCCESS)
19              printf("隐藏回收站成功");
20          RegCloseKey(hKey);               //关闭注册表项
21      }
22      else
23          printf("创建注册表值失败");
24  }
```

运行上面的代码，则会根据传入的参数显示或隐藏回收站。程序运行效果如图 21-12 所示，同时会隐藏回收站。

21.2.5　隐藏显示所有驱动器

要隐藏和显示驱动器，可以通过修改注册表实现。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Ex
plorer
```

在上面的注册表项中，建立名称为 NoDrives 的键。其值可以是多个驱动器的组合，如隐藏 A 盘，则此值为 1；隐藏 B 盘，此值为隐藏 A 盘的 2 倍，是 2；隐藏 C 盘，此值为隐藏 B 盘的 2 倍，是 4；依次类推。可以根据需要将需要隐藏的盘的值相加，如果要隐藏所有驱动器，则此值设置为 67108863。代码如下：

```
01  void HideDrivers(BOOL bHide)          //隐藏显示所有驱动器
02  {
03      DWORD dwValue;                    //定义设置的取值
04      if (bHide)
05          dwValue = 67108863;           //如果隐藏
06      else
07          dwValue = 0;                  //如果显示
08      HKEY hKey;                        //注册表键句柄
09      if (RegOpenKey(HKEY_CURRENT_USER,
10          "Software\\Microsoft\\Windows\\
11          CurrentVersion\\Policies\\Explorer",
12          &hKey) == ERROR_SUCCESS)      //创建注册表项
13      {
14          //设置注册表值
15          if(RegSetValueEx(hKey, "NoDrives", NULL, REG_DWORD,
16              (const BYTE*)&dwValue, sizeof(DWORD))
17              == ERROR_SUCCESS)
18              printf("隐藏所有驱动器成功");
19          RegCloseKey(hKey);            //关闭注册表项
20      }
21      else
22          printf("创建注册表值失败");
23  }
```

运行上面的代码，则会根据传入的参数隐藏或显示所有驱动器。程序运行效果如图 21-13 所示，同时会隐藏所有驱动器。

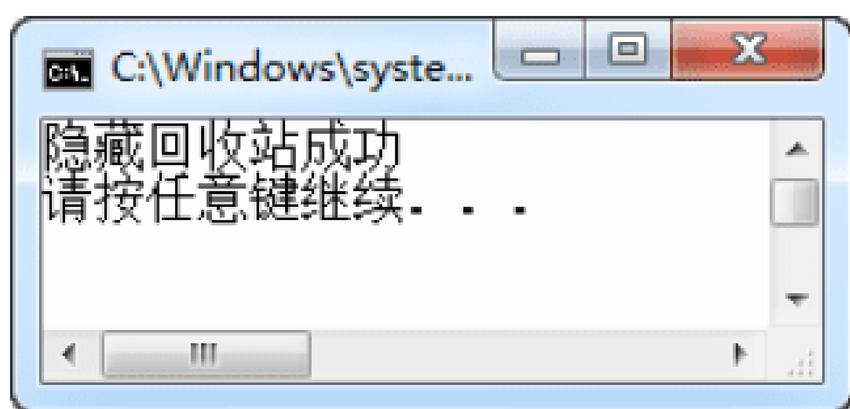


图 21-12　隐藏和显示回收站运行效果

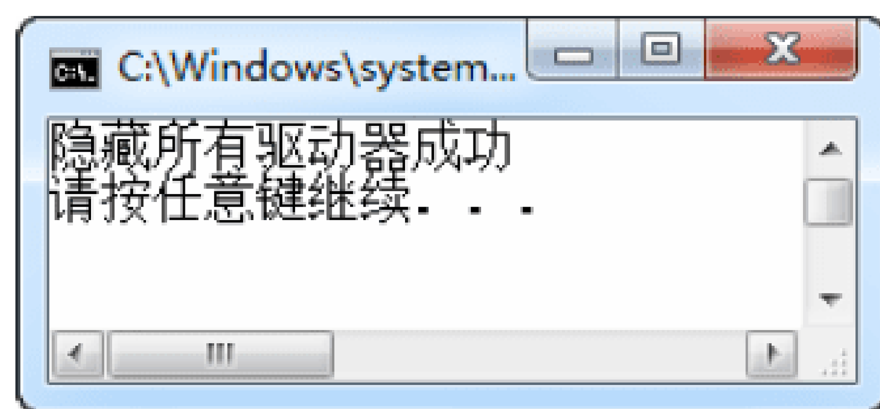


图 21-13　隐藏或显示所有驱动器运行效果

21.2.6　禁止“查找”菜单

要禁止“查找”菜单，可以通过修改注册表实现。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Ex
plorer
```

在上面的注册表项中，建立名称为 NoFind 的键，当值设置为 1 时，会禁止“查找”菜单。代码如下：

```
01  void DisableFindMenu()          //禁止“查找”菜单
02  {
03      DWORD dwValue=1;            //定义设置的取值
04      HKEY hKey;                  //注册表键句柄
05      if (RegOpenKey(HKEY_CURRENT_USER,
06          "Software\\Microsoft\\Windows\\
07          CurrentVersion\\Policies\\Explorer",
08          &hKey) == ERROR_SUCCESS) //打开注册表项
09      {
10          //设置注册表值
11          if(RegSetValueEx(hKey, "NoFind", NULL, REG_DWORD,
12              (const BYTE*)&dwValue, sizeof(DWORD)) == ERROR_SUCCESS)
13              printf("禁止查找菜单成功");
14          else
15              printf("禁止查找菜单失败");
16          RegCloseKey(hKey);          //关闭注册表项
17      }
18      else
19          printf("打开注册表值失败");
20  }
```

运行上面的代码，则会禁止“查找”菜单。程序运行效果如图 21-14 所示，同时会禁止“查找”菜单。

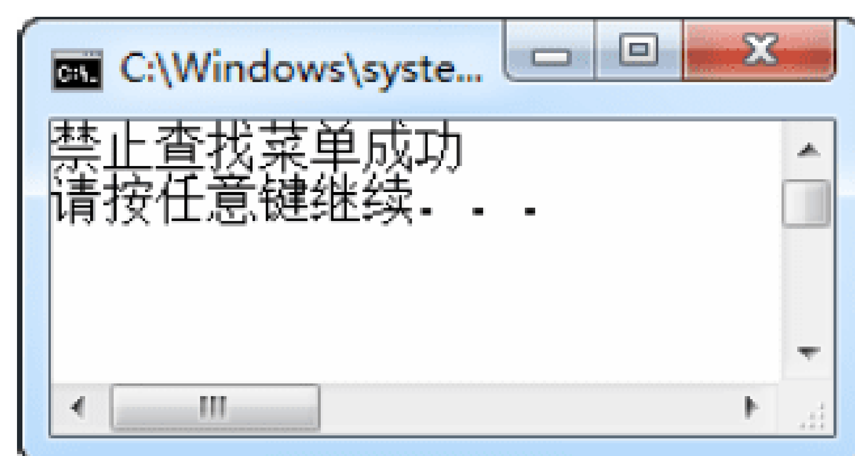


图 21-14　禁止“查找”菜单运行效果

21.2.7　禁止“文档”菜单

要禁止“文档”菜单，可以通过修改注册表实现。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Ex
plorer
```

在上面的注册表项中，建立名称为 NoRecentDocsMenu 的键，其值设置为 1 时，会禁止“文档”菜单。代码如下所示：

```
01  void DisableDocumentMenu()       //禁止“文档”菜单
02  {
03      DWORD dwValue=1;             //定义设置的取值
04      HKEY hKey;                   //注册表键句柄
05      if (RegOpenKey(HKEY_CURRENT_USER,
06          "Software\\Microsoft\\Windows\\
07          CurrentVersion\\Policies\\Explorer",
08          &hKey) == ERROR_SUCCESS) //打开注册表项
09      {
10          //设置注册表值
11          if(RegSetValueEx(hKey, "NoRecentDocsMenu", NULL, REG_DWORD,
12              (const BYTE*)&dwValue, sizeof(DWORD)) == ERROR_SUCCESS)
13              printf("禁止文档菜单成功");
14          else
15              printf("禁止文档菜单失败");
16          RegCloseKey(hKey);       //关闭注册表项
17      }
18      else
19          printf("打开注册表值失败");
20  }
```

运行上面的代码，则会禁止“文档”菜单。程序运行效果如图 21-15 所示，同时会禁止“文档”菜单。

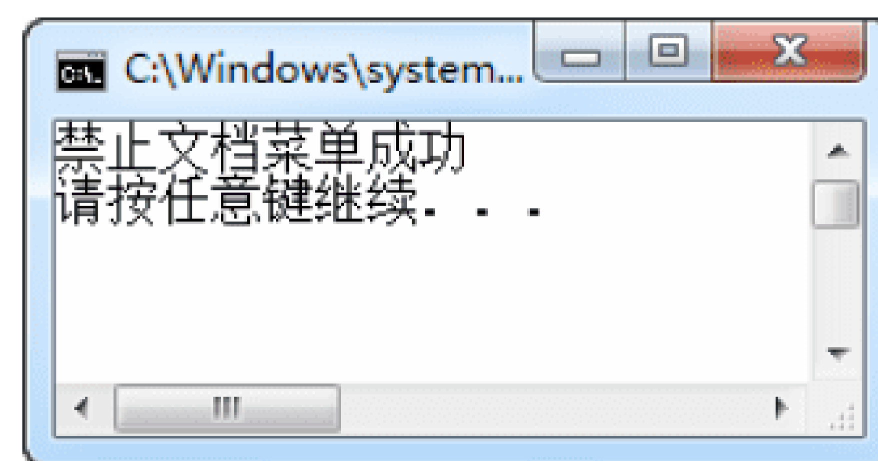


图 21-15　禁止“文档”菜单运行效果

21.2.8　在退出 Windows 时清除“文档”中的记录

要在退出 Windows 时清除“文档”中的记录，可以通过修改注册表实现。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Ex
plorer
```

在上面的注册表项中，建立名称为 ClearRecentDocsonExit 的键，其值设置为二进制{01,00,00,00}时，会在退出 Windows 时清除“文档”中的记录。代码如下：

```
01  void ClearRecentDocBeforeExit()//在退出 Windows 时清除“文档”中的记录
02  {
03      BYTE szValue[4]={01,00,00,00};  //定义设置的取值
04      HKEY hKey;                      //注册表键句柄
05      if (RegOpenKey(HKEY_CURRENT_USER,
06          "Software\\Microsoft\\Windows\\
07          CurrentVersion\\Policies\\Explorer",
08          &hKey) == ERROR_SUCCESS)    //打开注册表项
09      {
10          if(RegSetValueEx(hKey, "ClearRecentDocsonExit", NULL,
11              REG_BINARY,(const BYTE*)&szValue,
12              sizeof(szValue)) == ERROR_SUCCESS)
13              printf("在退出 WINDOWS 时清除文档中的记录 成功");
14          else
15              printf("在退出 WINDOWS 时清除文档中的记录 失败");
```

```
16          RegCloseKey(hKey);
17      }
18      else
19          printf("打开注册表值失败");
20  }
```

运行上面的代码，则会在退出 Windows 时清除“文档”中的记录。程序运行效果如图 21-16 所示。

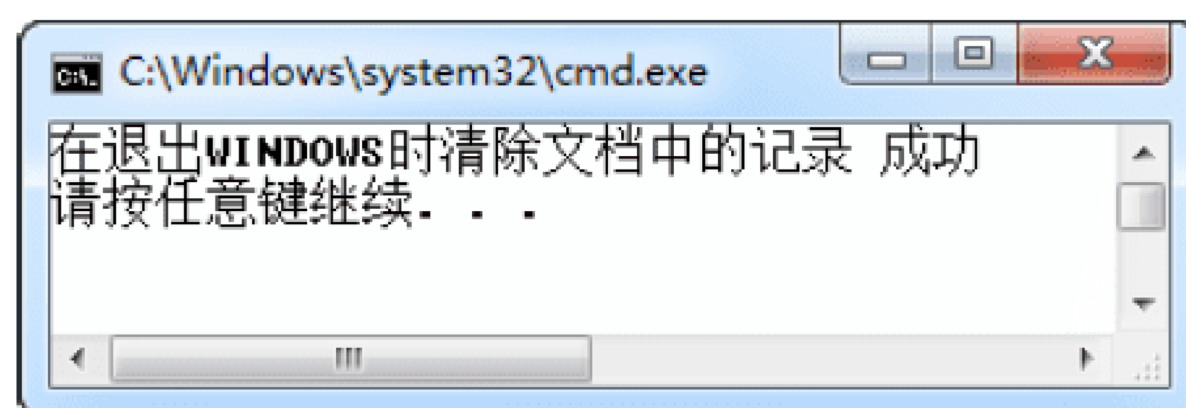


图 21-16　退出 Windows 时清除“文档”中的记录运行效果

21.2.9　禁用注册表编辑器

要禁用注册表编辑器，可以通过修改注册表实现。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\System
```

在上面的注册表项中，建立名称为 DisableRegistryTools 的键，其值设置为 1 时，会禁用注册表编辑器；其值设置为 0 时，会启用注册表编辑器。代码如下：

```
01  void DisableRigisterEdit(BOOL bHide)//禁用注册表编辑器
02  {
03      DWORD dwValue;                      //定义设置的取值
04      if (bHide)
05          dwValue = 1;                    //如果隐藏
06      else
07          dwValue = 0;                    //如果显示
08      HKEY hKey;                          //注册表键句柄
09      if (RegCreateKey(HKEY_CURRENT_USER,
10          "Software\\Microsoft\\Windows\\
11          CurrentVersion\\Policies\\System",
12          &hKey) == ERROR_SUCCESS)        //打开注册表项
13      {
14          //设置注册表值
15          if(RegSetValueEx(hKey,"DisableRegistryTools",NULL,REG_DWORD,
16              (const BYTE*)&dwValue, sizeof(DWORD)) == ERROR_SUCCESS)
17              printf("禁用注册表编辑器成功");
18          else
19              printf("禁用注册表编辑器失败");
20          RegCloseKey(hKey);
21      }
22      else
23          printf("打开注册表值失败");
24  }
```

运行上面的代码，则会禁用注册表编辑器。程序运行效果如图 21-17 所示，同时会禁用注册表编辑器。

21.2.10　禁止使用 inf 文件

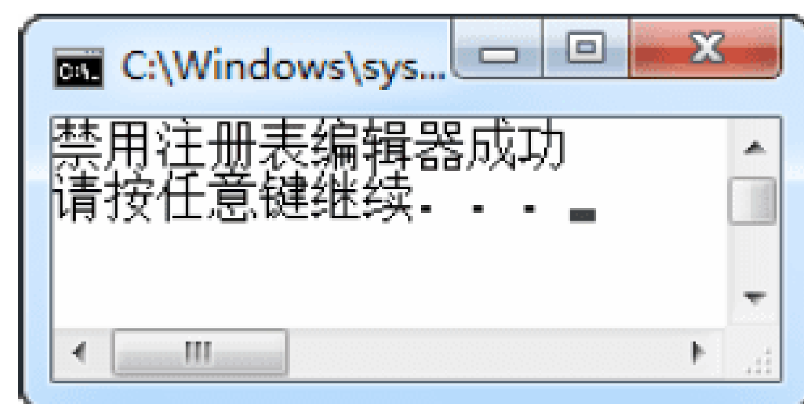


图 21-17　禁用注册表编辑器运行效果

操作系统在 HKEY_CLASSES_ROOT 注册表根项下，注册所有使用的文件类型，以其扩展名前加点号为注册表项的名称，如 inf 文件的配置项存放在 HKEY_CLASSES_ROOT\.inf 注册表项中，其默认值为 inffile。如果修改此默认值，则系统将禁用 inf 文件。代码如下：

```
01  void DisableInfFile(BOOL bDisable)                  //禁止使用 inf 文件
02  {
03      char szValue[50]={0};                           //定义设置的取值
04      if (bDisable)
05          strcpy(szValue, "txtfile");                 //如果禁用 inf 文件
06      else
07          strcpy(szValue, "inffile");                 //如果启用 inf 文件
08      HKEY hKey;                                      //注册表键句柄
09      if (RegOpenKey(HKEY_CLASSES_ROOT, ".inf",
10          &hKey) == ERROR_SUCCESS)                    //打开注册表项
11      {                                               //设置注册表值
12          if(RegSetValueEx(hKey, NULL,NULL,REG_SZ,(const BYTE*)&szValue,
13              strlen(szValue)) == ERROR_SUCCESS)
14              printf("禁止使用 inf 文件成功");
15          else
16              printf("禁止使用 inf 文件失败");
17          RegCloseKey(hKey);                          //关闭注册表项
18      }
19      else
20          printf("打开注册表值失败");
21  }
```

运行上面的代码，则会禁止使用 inf 文件。程序运行效果如图 21-18 所示，同时会禁止使用 inf 文件。

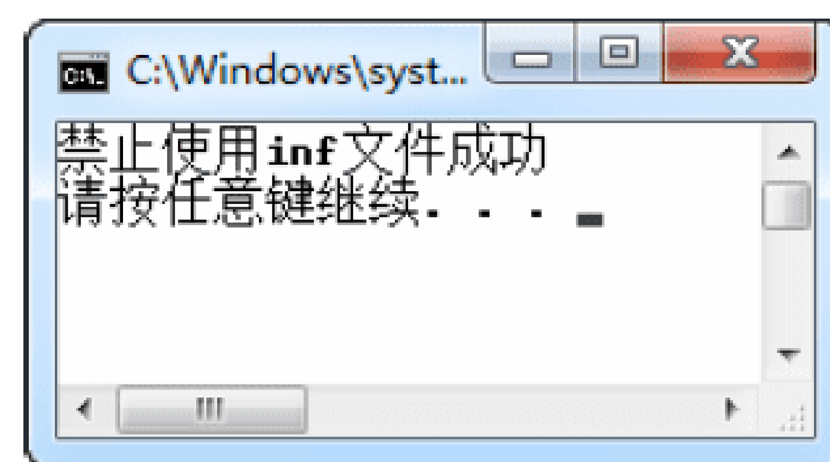


图 21-18　禁止使用 inf 文件运行效果

21.2.11　禁止使用 reg 文件

与 inf 文件一样，reg 文件的配置项存放在 HKEY_CLASSES_ROOT\.reg 注册表项中，默认值为 regfile。如果修改此默认值，则系统将禁用 reg 文件。代码如下：

```
01  void DisableRegFile(BOOL bDisable)                  //禁止使用 reg 文件
02  {
03      char szValue[50]={0};                           //定义设置的取值
04      if (bDisable)
05          strcpy(szValue, "txtfile");                 //如果禁用 reg 文件
06      else
07          strcpy(szValue, "regfile");                 //如果启用 reg 文件
08      HKEY hKey;                                      //注册表键句柄
09      if (RegOpenKey(HKEY_CLASSES_ROOT, ".reg",
10          &hKey) == ERROR_SUCCESS)                    //打开注册表项
11      {                                               //设置注册表值
```

```
12          if(RegSetValueEx(hKey,NULL,NULL,REG_SZ,(const BYTE*)&szValue,
13              strlen(szValue)) == ERROR_SUCCESS)
14              printf("禁止使用 reg 文件成功");
15          else
16              printf("禁止使用 reg 文件失败");
17          RegCloseKey(hKey);                              //关闭注册表项
18      }
19      else
20          printf("打开注册表值失败");
21  }
```

运行上面的代码，则会禁止使用 reg 文件。程序运行效果如图 21-19 所示，同时会禁止使用 reg 文件。

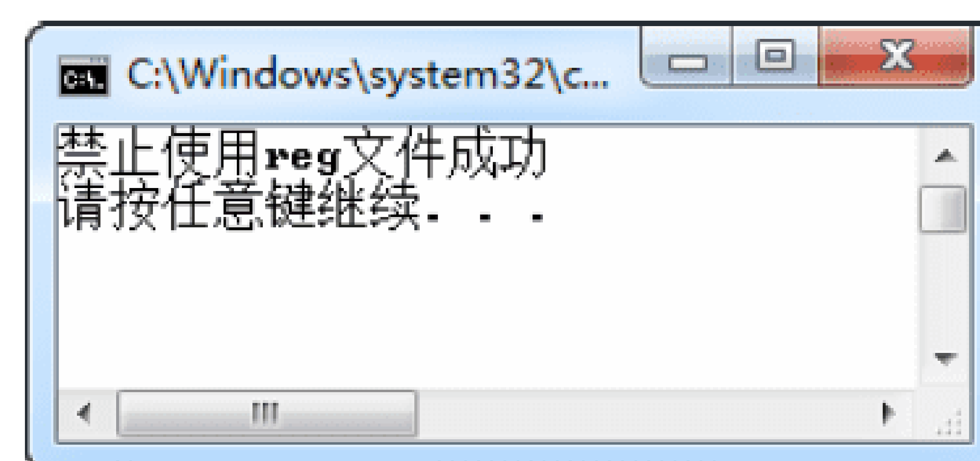


图 21-19　禁止使用 reg 文件运行效果

21.2.12　显示隐藏文件或文件夹

通过修改注册表可以显示隐藏文件或文件夹。其在如下注册表项中实现：

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Ex
plorer\AdvancedFolderHiddenSHOWALL
```

在上面的注册表项中，建立名称为 CheckedValue 的键，其值设置为 1 时，会禁止显示隐藏文件或文件夹；其值设置为 0 时，允许显示隐藏文件或文件夹。代码如下：

```
01  void ShowHideFile(BOOL bDisable)          //显示隐藏文件或文件夹
02  {
03      DWORD dwValue;                         //定义设置的取值
04      if (bDisable)
05          dwValue = 1;                       //如果隐藏
06      else
07          dwValue = 0;                       //如果显示
08      HKEY hKey;                             //注册表键句柄
09      if (RegCreateKey(HKEY_CURRENT_USER,
10          "Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\
11          Explorer\\AdvancedFolderHiddenSHOWALL",
12          &hKey) == ERROR_SUCCESS)           //创建注册表项
13      {
14          //设置注册表值
15          if(RegSetValueEx(hKey, "CheckedValue", NULL, REG_DWORD,
16              (const BYTE*)&dwValue, sizeof(DWORD)) == ERROR_SUCCESS)
17              printf("显示隐藏文件或文件夹成功");
18          else
19              printf("显示隐藏文件或文件夹失败");
20          RegCloseKey(hKey);         //关闭注册表项
21      }
22      else
23          printf("创建注册表值失败");
24  }
```

运行上面的代码，则会显示隐藏文件或文件夹。程序运行效果如图 21-20 所示，同时会显示隐藏文件或文件夹。

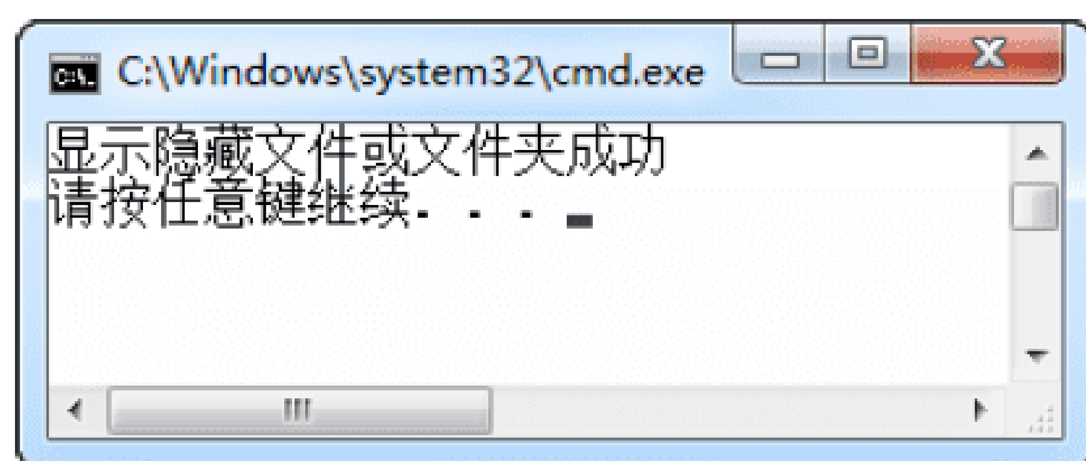


图 21-20　显示隐藏文件或文件夹运行效果

21.3　读写注册表的 ATL 类

MFC 中封装了 CRegKey 类实现对系统注册表中的值的操作，提供对系统注册表操作的编程接口。在使用 CRegKey 类时，需要加入对 atlbase.h 头文件的引用。本节将介绍使用此类对注册表键值进行读写操作的方法。

21.3.1　使用 CRegKey 类写入默认键值

在 CRegKey 类中，要操作注册表键，首先需要创建或打开注册表键，使用 Create()函数可以打开或创建注册表键。如果指定的注册表键在父键下存在，则打开注册表键，否则在父键下创建新键。其函数原型为：

```
LONG Create(
  HKEY hKeyParent,       //表示打开的键的句柄
  LPCTSTR lpszKeyName,  //指定要创建或打开的键名，此名称为 hKeyParent 的子键
  LPTSTR lpszClass = REG_NONE,                    //指定要创建或打开的键的类
  DWORD dwOptions = REG_OPTION_NON_VOLATILE,    //打开或创建键的模式
  REGSAM samDesired = KEY_ALL_ACCESS,
                        //指定键的访问安全性，默认值表示完成所有的操作
  LPSECURITY_ATTRIBUTES lpSecAttr = NULL,        //创建参数
  LPDWORD lpdwDisposition = NULL );
                        //输出参数，返回操作是创建新键值还是打开已有的键值
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则，返回错误值。打开注册表键后，就可以读写注册表键，使用 CRegKey 类的 SetValue()函数可以为指定键值写入值。其函数原型为：

```
LONG SetValue(
  DWORD dwValue,     //指定要设置的整型值
  LPCTSTR lpszValueName );
                    //指定要设置的值的名称，如果此名称的值域不存在，则会添加
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则，返回错误值。结合上面这两个函数，就可以使用 CregKey 类写入默认键值，代码如下：

```
01  //写入默认值函数测试
02  void CCRegKeySampleDlg::OnButtonWritedefaultkey()
03  {
04      CRegKey key;                                //定义 CRegKey 变量
05      long result = key.Create(HKEY_CURRENT_USER,
06          "MyTestKey\\Animals\\Pig");
```

```
07      if (result != ERROR_SUCCESS)                    //如果失败，则提示
08      {
09          MessageBox("创建注册表键失败","错误");
10          return;
11      }
12      //写入默认值
13      result = key.SetValue("体态肥胖，好吃懒做，但是活泼可爱");
14      if (result != ERROR_SUCCESS)
15          MessageBox("写入注册表默认键值失败", "错误");
16      else
17          MessageBox("写入注册表默认键值成功", "提示");
18  }
```

上面代码首先使用 CRegKey 类的 Create()函数创建或打开注册表键。如果已经存在指定的键，则打开，如果不存在，则创建相应的键。然后使用 CRegKey 类的 SetValue()函数设置指定键的默认值域的值，此处为“体态肥胖，好吃懒做，但是活泼可爱”。从上面可以看出使用 SetValue 可以不需要打开注册表键而直接写入键值，这里不再赘述。程序运行后，注册表效果如图 21-21 所示。

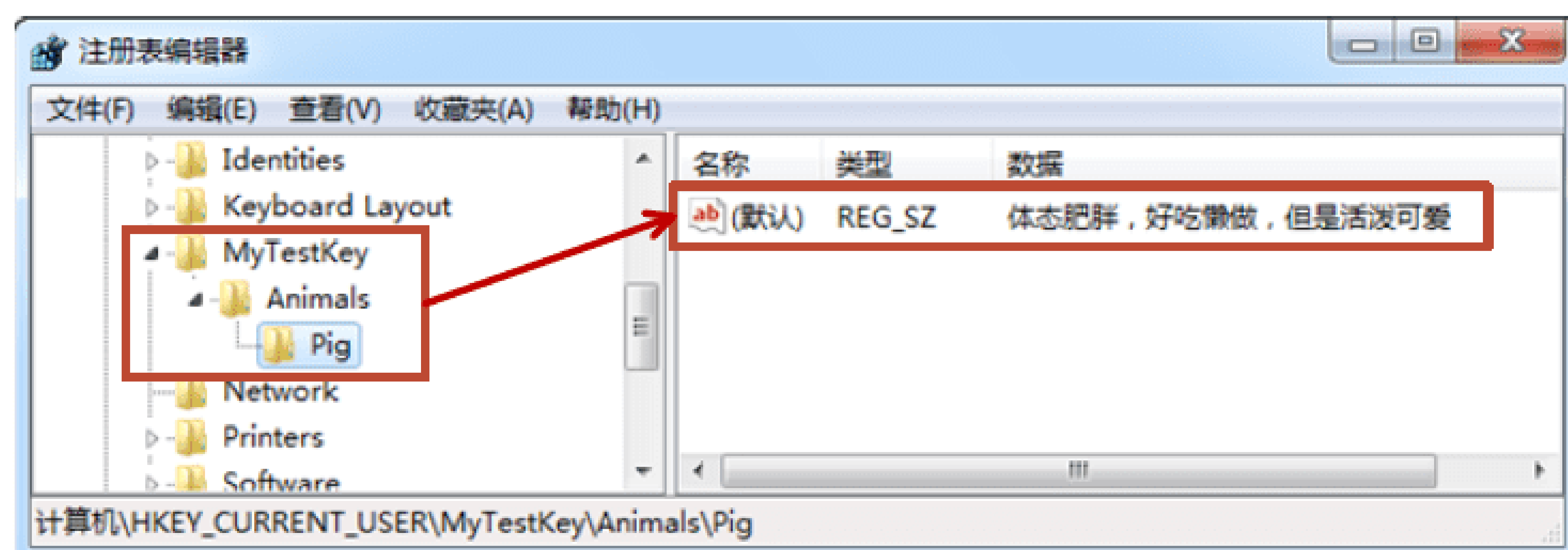


图 21-21　写入默认键值运行效果

21.3.2　使用 CRegKey 类写入新键值

除了写入注册表键的默认键值外，还可以为其写入新键值。使用 CRegKey 的 SetKeyValue()函数创建或打开键名，并为指定名称的值域写入存储值。其函数原型为：

```
LONG SetKeyValue(
  LPCTSTR lpszKeyName,                    //指定要创建或打开的键的名称
  LPCTSTR lpszValue,                      //指定要设置的值
  LPCTSTR lpszValueName = NULL );         //指定要设置的值的名称
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则，返回错误值。下面是使用 SetKeyValue()函数写入新键值的代码。

```
01  void CCRegKeySampleDlg::OnButtonWritenewkey()       //写入新键值
02  {
03      CRegKey key;                                     //CRegKey 变量
04      long result1;                                    //结果长整型
05      //创建键值
06      long result=key.Create(HKEY_CURRENT_USER,"MyTestKey\\Animals\\");
07      if (result != ERROR_SUCCESS)                     //如果失败则提示
08      {
09          MessageBox("创建注册表键失败", "错误");
```

```
10          return;
11        }
12        result1 = key.SetKeyValue("Pig", "好吃懒做，但是活泼可爱", "特点");
13        //设置键值
14        result = key.SetKeyValue("Pig", "肥胖", "体态"); //设置键值
15        //提示操作结果
16        if ((result != ERROR_SUCCESS) || (result1 != ERROR_SUCCESS))
17          MessageBox("写入注册表新键值失败", "错误");
18        else
19          MessageBox("写入注册表新键值成功", "提示");
20    }
```

上面代码首先使用 CRegKey 类的 Create()函数创建或打开注册表键。如果已经存在指定的键，则打开，如果不存在，则创建相应的键。然后使用 CRegKey 类的 SetKeyValue()函数设置指定键的指定值域的值，此处添加值域为“特点”的值为“好吃懒做，但是活泼可爱”和值域为“体态”的值为“肥胖”。程序运行后，注册表效果如图 21-22 所示。

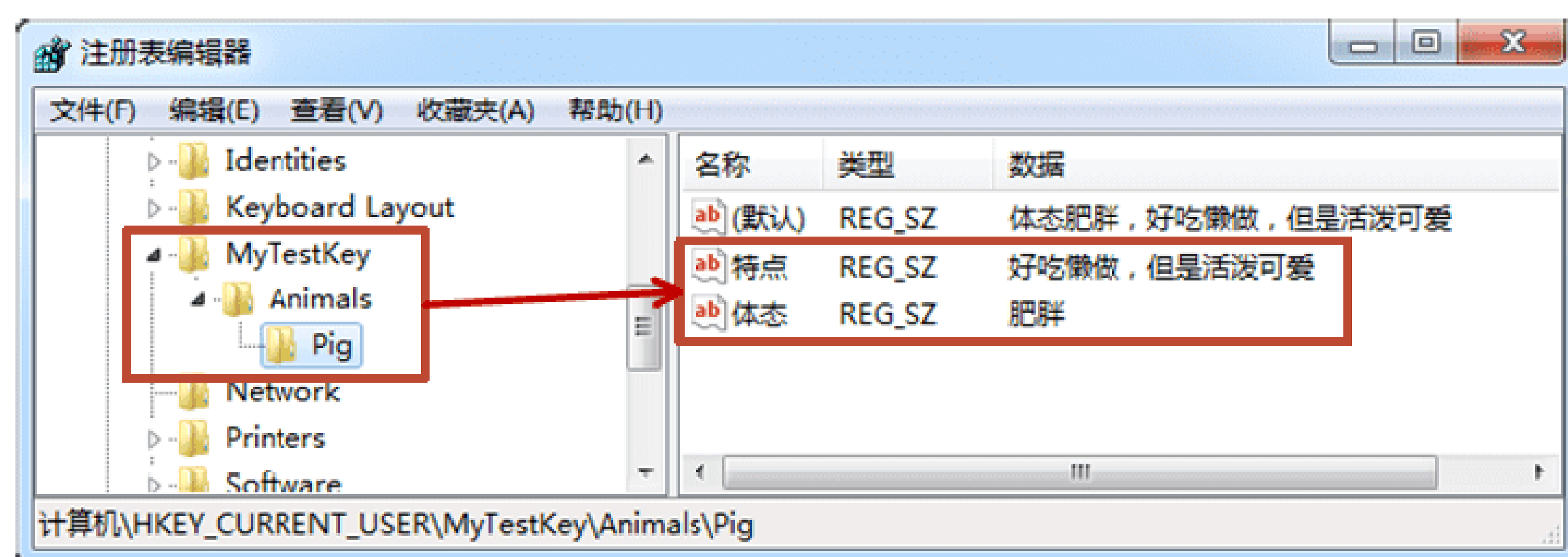


图 21-22　写入新键值运行效果

21.3.3　使用 CRegKey 类查询键值

使用 CregKey 类的 QueryValue()函数可以查询注册表键值。其函数原型为：

```
LONG QueryValue(
  DWORD& dwValue,                    //存储查询到的整型值
  LPCTSTR lpszValueName );           //指定要查询的值域的名称
LONG QueryValue(
  LPTSTR szValue,                    //存储查询到的字符串值
  LPCTSTR lpszValueName,             //指定要查询的值域的名称
  DWORD* pdwCount )                  //返回查询到的字符串的长度
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则，返回错误值。下面是使用 QueryValue()函数查询注册表键值的代码。

```
01    void CCRegKeySampleDlg::OnButtonQuerykey()  //查询键值
02    {
03        CRegKey key;                                    //CRegKey 变量
04        //创建键值
05        long result = key.Create(HKEY_CURRENT_USER,
06                                "MyTestKey\\Animals\\Pig");
07        if (result != ERROR_SUCCESS)                   //如果失败，则提示
08        {
09          MessageBox("打开注册表键失败", "错误");
10          return;
```

```
11      }
12      char chat[256];                         //存放返回的值
13      DWORD iLen = 256;                       //存放取值长度
14      result = key.QueryValue(chat, "体态", &iLen);//查询键值
15      //提示信息
16      if (result != ERROR_SUCCESS)
17          MessageBox("查询注册表键失败", "错误");
18      else
19          MessageBox(chat, "Pig 的体态是");
20  }
```

上面代码首先使用 CRegKey 类的 Create()函数创建或打开注册表键。如果已经存在指定的键，则打开，如果不存在，则创建相应的键。然后使用 CregKey 类的 QueryValue()函数查询指定键的指定值域的值，此处查询值域为“体态”的值。程序运行效果如图 21-23 所示。



图 21-23　查询键值运行效果

21.4　注册表的查询与枚举

前面几节介绍了注册表的读写方法和有关注册表的应用。本节将介绍如何实现注册表的查询和枚举，使用这两个功能，可以完成对注册表的管理。本节具体介绍查询注册表键值的方法、枚举注册表项的方法、枚举注册表键值的方法以及枚举注册表项和注册表键值的应用——列举注册表中的启动项和枚举安装程序。

21.4.1　查询注册表键值

使用 RegQueryValueEx()函数可以查询指定注册表键值的信息。此函数可以获取已经打开的注册表项的指定名称的键的值和数据类型。使用此函数查询前，需要先打开注册表项，查询完数据后，需要调用 RegCloseKey()函数关闭对注册表项的访问。其函数原型为：

```
LONG RegQueryValueEx(
 HKEY hKey,                         //要查询键值所在的注册表项的句柄
 LPTSTR lpValueName,                //表示要查询的注册表项的键值
 LPDWORD lpReserved,                //预留参数
 LPDWORD lpType,                    //用于获取键值的数据类型
 LPBYTE lpData,                     //用于存放返回的注册表键值
 LPDWORD lpcbData );                //用于存放返回的注册表键值的数据长度
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则返回非 0。下面代码显示了如何查询注册表键值的信息。

```
01  void QueryKeyValue()                            //查询注册表键值信息
02  {
03      DWORD szValue;                              //定义取值变量
04      DWORD dwType;                               //定义类型变量
05      DWORD dwLength = sizeof(szValue);           //获取值变量长度
06      HKEY hKey;                                  //注册表项句柄
07      if (RegOpenKey(HKEY_LOCAL_MACHINE,
08          "Software\\LLN\\VC\\Registry\\CreateTest",
09          &hKey) == ERROR_SUCCESS)                //打开注册表
10      {
11          if(RegQueryValueEx(hKey, NULL,NULL,&dwType,(LPBYTE)&szValue,
12              &dwLength) == ERROR_SUCCESS)        //查询注册表键值
13          {
14              switch(dwType)    //根据类型，分析判断返回的注册表键值的数据类型
15              {
16              case REG_BINARY:
17                  break;
18              case REG_DWORD_BIG_ENDIAN:
19              case REG_DWORD:
20                  printf("成功.类型=REG_DWORD;键值=%u", UINT(szValue));
21                  break;
22              case REG_EXPAND_SZ:
23              case REG_MULTI_SZ:
24              case REG_SZ:
25                  printf("成功.类型=REG_SZ;键值=%s",szValue);
26                  break;
27              case REG_LINK:
28                  printf("成功.类型=REG_LINK;");
29                  break;
30              case REG_RESOURCE_LIST:
31                  printf("成功.类型=REG_RESOURCE_LIST;");
32                  break;
33              case REG_NONE:
34                  printf("成功.类型=REG_NONE;");
35                  break;
36              default:
37                  printf("成功.类型=未知;");
38                  break;
39              }
40          }
41          RegCloseKey(hKey);                      //关闭注册表句柄
42      }
43      return;
44  }
```

上面代码调用 RegOpenKey()函数打开注册表项后，调用 RegQueryValueEx()函数查询注册表项的默认键值，并判断返回的键值的类型和取值，并显示出来，最后调用 RegCloseKey()函数关闭注册表句柄。程序运行效果如图 21-24 所示。

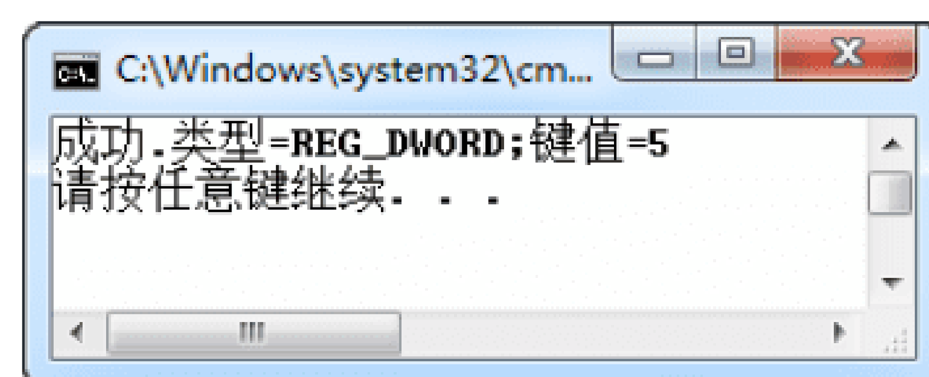


图 21-24　查询注册表键值运行效果

21.4.2　快速查询注册表键值

使用 RegQueryValue()函数可以快速查询指定注册表键值的信息，而不需要打开和关闭

注册表项句柄。其函数原型为：

```
LONG RegQueryValue(
  HKEY hKey,                          //要查询键值所在的注册表项的句柄
  LPCTSTR lpSubKey,                   //表示要查询的注册表项的路径
  LPTSTR lpValue,                     //用于存放返回的注册表键值的数据
  PLONG lpcbValue );                  //用于存放返回的注册表键值的数据长度
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则返回非 0。下面代码显示了快速查询注册表键值的方法。

```
01  void QuickQueryKeyValue()                              //快速查询注册表键值
02  {
03      char szValue[MAX_PATH];                            //取值变量
04      long dwLength = sizeof(szValue);                   //取值长度
05      if (RegQueryValue(HKEY_LOCAL_MACHINE,
06          "Software\\LLN\\VC\\Registry\\CreateTest",
07          (char*)&szValue, &dwLength) == ERROR_SUCCESS)//查询键值
08          printf("成功.键值=%s",szValue);                //输出键值
09  }
```

上面代码使用 RegQueryValue()函数直接查询注册表项 HKEY_LOCAL_MACHINE \Software\LLN\VC\Registry\CreateTest 的默认键值，并在屏幕上显示该值。程序运行效果如图 21-25 所示。

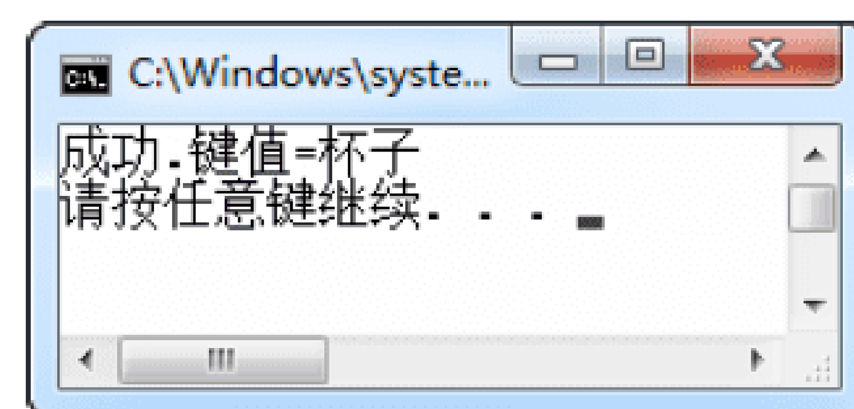


图 21-25　快速查询注册表键值运行效果

21.4.3　枚举注册表键值

使用 RegEnumKeyEx()函数可以枚举注册表键值。每调用一次函数会返回一个键值。使用此函数枚举前，需要先打开注册表项，枚举完数据后，再调用 RegCloseKey()函数关闭对注册表项的访问。其函数原型为：

```
LONG RegEnumValue(
  HKEY hKey,                    //要枚举键值的注册表项的句柄
  DWORD dwIndex,                //表示要枚举的注册表键值的索引
  LPTSTR lpValueName,           //用于存放返回的注册表键值的名称
  LPDWORD lpcbValueName,        //用于存放返回的注册表键值名称的长度
  LPDWORD lpReserved,           //预留参数
  LPDWORD lpType,               //用于存放返回的注册表键值的数据类型
  LPBYTE lpData,                //用于存放返回的注册表键值中的数据
  LPDWORD lpcbData);            //用于存放返回的注册表键值中的数据长度
```

如果函数成功，则返回 ERROR_SUCCESS；否则返回非 0。

21.4.4　列举开机启动程序

21.4.3 小节介绍了枚举注册表键值的 API 函数。本小节结合一个实例，讲解如何使用这个函数。Windows 操作系统中的开机启动项存放在 HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run 注册表项中，通过枚举此注册表项下的键值，就可以列举出 Windows 操作系统中的开机启动项。代码如下：

```
01  void EnumStartProgram()                          //枚举开机启动项
02  {
03      HKEY hKey;                                   //注册表键值
04      DWORD dwIndex=0;                             //索引值
05      char szValueName[MAX_PATH]={0};              //取值变量
06      DWORD dwValueName=sizeof(szValueName);       //取值长度
07      DWORD dwType;                                //类型值
08      BYTE szData[MAX_PATH]={0};                   //存放数据的变量
09      DWORD dwData=sizeof(szData);                 //数据区长度
10      //打开注册表项
11      if (RegOpenKey(HKEY_CURRENT_USER,
12          "Software\\Microsoft\\Windows\\CurrentVersion\\Run",
13          &hKey) == ERROR_SUCCESS)
14      {
15          //循环枚举注册表键值
16          while (((RegEnumValue(hKey, dwIndex, (LPTSTR)szValueName,
17              &dwValueName,NULL, &dwType, (LPBYTE)szData,
18              &dwData)) == ERROR_SUCCESS))
19          {
20              printf("[%d]%s=%sd\n", dwIndex, szValueName, szData);
21              //输出键值
22              dwIndex++;                           //增加索引
23              dwValueName=sizeof(szValueName); //获取键名称长度
24              memset(szValueName,0,dwValueName);//初始化键名称变量
25              dwData=sizeof(szData);               //获取数据区长度
26              memset(szData,0,dwData);             //初始化数据区长度
27          }
28          RegCloseKey(hKey);                       //关闭注册表句柄
29      }
30  }
```

上面的代码首先使用 RegOpenKey()函数打开包含系统启动项的注册表项，然后使用 while 循环调用 RegEnumValue()函数枚举每个键值，索引从 0 开始，每次增 1，直到 RegEnumValue()函数返回值不为 ERROR_SUCCESS，表明枚举结束。在调用 RegEnumValue()函数成功后，程序会显示出键值名称和键值数据。程序运行的效果如图 21-26 所示。

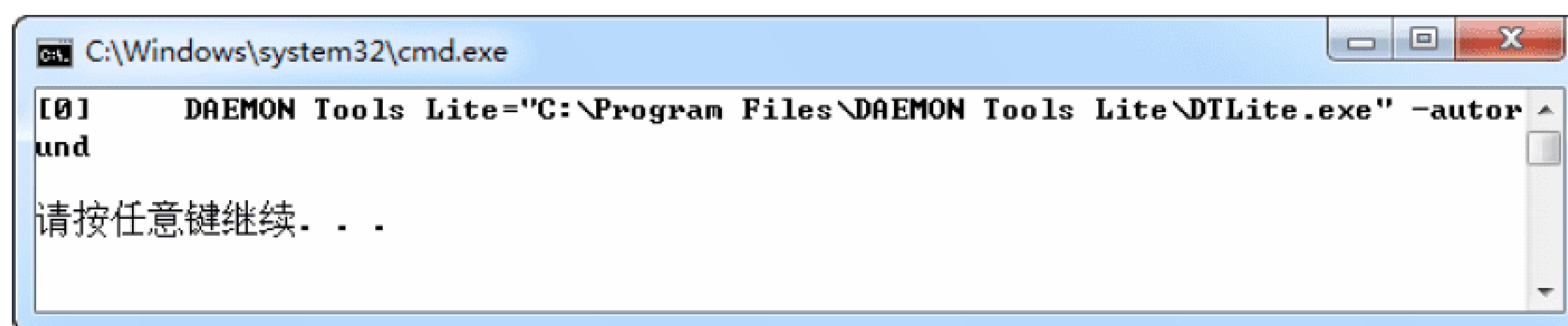


图 21-26　列举注册表中启动项运行效果

21.4.5　枚举注册表项

使用 RegEnumKey()函数可以枚举指定注册表项的子项。每调用一次函数会返回一个子项。使用此函数枚举前，需要先打开注册表项，枚举完数据后，需要调用 RegCloseKey()函数关闭对注册表项的访问。其函数原型为：

```
LONG RegEnumKey(
  HKEY hKey,                             //要枚举子项的注册表项的句柄
```

```
  DWORD dwIndex,                    //要枚举的注册表项的子项的索引
  LPTSTR lpName,                    //用于存放返回的注册表子项的名称
  DWORD cbName);                    //用于存放返回的注册表子项的名称的长度
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则返回非 0。除了此 API 函数外，系统还提供 RegEnumKeyEx()函数实现更丰富的枚举注册表子项的功能，不仅可以获取注册表子项名称，还可以获取包括子项类名和最后一次修改时间的信息。其函数原型为：

```
LONG RegEnumKeyEx(
  HKEY hKey,                    //要枚举子项的注册表项的句柄
  DWORD dwIndex,                //表示要枚举的注册表项的子项的索引
  LPTSTR lpName,                //用于存放返回的注册表子项的名称
  LPDWORD lpcbName,             //用于存放返回的注册表子项的名称的长度
  LPDWORD lpReserved,           //预留参数
  LPTSTR lpClass,               //用于存放返回的注册表子项的类名
  LPDWORD lpcbClass,            //用于存放返回的注册表子项的类名的长度
  PFILETIME lpftLastWriteTime );//用于存放注册表子项最后一次修改的时间
```

如果函数操作成功，则返回 ERROR_SUCCESS；否则返回非 0。

21.4.6　枚举安装程序

21.4.5 小节介绍了枚举注册表项的两个 API 函数。本小节结合一个实例，讲解如何使用这两个函数。Windows 操作系统中所有安装程序的信息存放在 HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\WINDOWS\CurrentVersion\uninstall 注册表项中，通过枚举此注册表项下的子项，就可以枚举出所有的安装程序。代码如下：

```
01  void EnumKeyProgram()                               //枚举安装的应用程序
02  {
03      HKEY hKey;                                      //注册表句柄
04      DWORD dwIndex=0;                                //索引值
05      char szKeyName[MAX_PATH]={0};                   //键名称变量
06      DWORD dwNameLen=sizeof(szKeyName);              //键名称变量
07      char szKeyClass[MAX_PATH]={0};                  //类型变量
08      DWORD dwClassLen=sizeof(szKeyClass);            //类型长度
09      FILETIME item={0};                              //文件项变量
10      if (RegOpenKey(HKEY_LOCAL_MACHINE,
11          "SOFTWARE\\MICROSOFT\\WINDOWS\\CurrentVersion\\uninstall",
12          &hKey) == ERROR_SUCCESS)                    //打开注册表句柄
13      {
14          while (((RegEnumKeyEx(hKey, dwIndex, szKeyName, &dwNameLen,
15              NULL, szKeyClass, &dwClassLen, &item)) == ERROR_SUCCESS))
16          {
17              //循环枚举注册表键
18              printf("[%d]名称=%sd\n", dwIndex, szKeyName);
19              dwIndex++;                              //增加索引
20              dwNameLen=sizeof(szKeyName);            //获取键名长度
21              memset(szKeyName,0,dwNameLen);          //初始化键名
22              dwClassLen=sizeof(szKeyClass);          //获取类型长度
23              memset(szKeyClass,0,dwClassLen);        //初始化类名
24              memset(&item,0,sizeof(item));           //初始化文件项
25          }
```

```
26          RegCloseKey(hKey);                              //关闭注册表句柄
27      }
28  }
```

上面的代码首先使用 RegOpenKey()函数打开包含安装程序信息的注册表项，然后使用 while 循环调用 RegEnumKeyEx()函数枚举每一个安装的程序，索引从 0 开始，每次增 1，直到 RegEnumKeyEx()函数返回值不为 ERROR_SUCCESS，表明枚举结束。在调用 RegEnumKeyEx()函数成功后，程序会显示出程序名称，读者可以根据需要，读取类名和最近一次修改时间。程序运行的效果如图 21-27 所示。

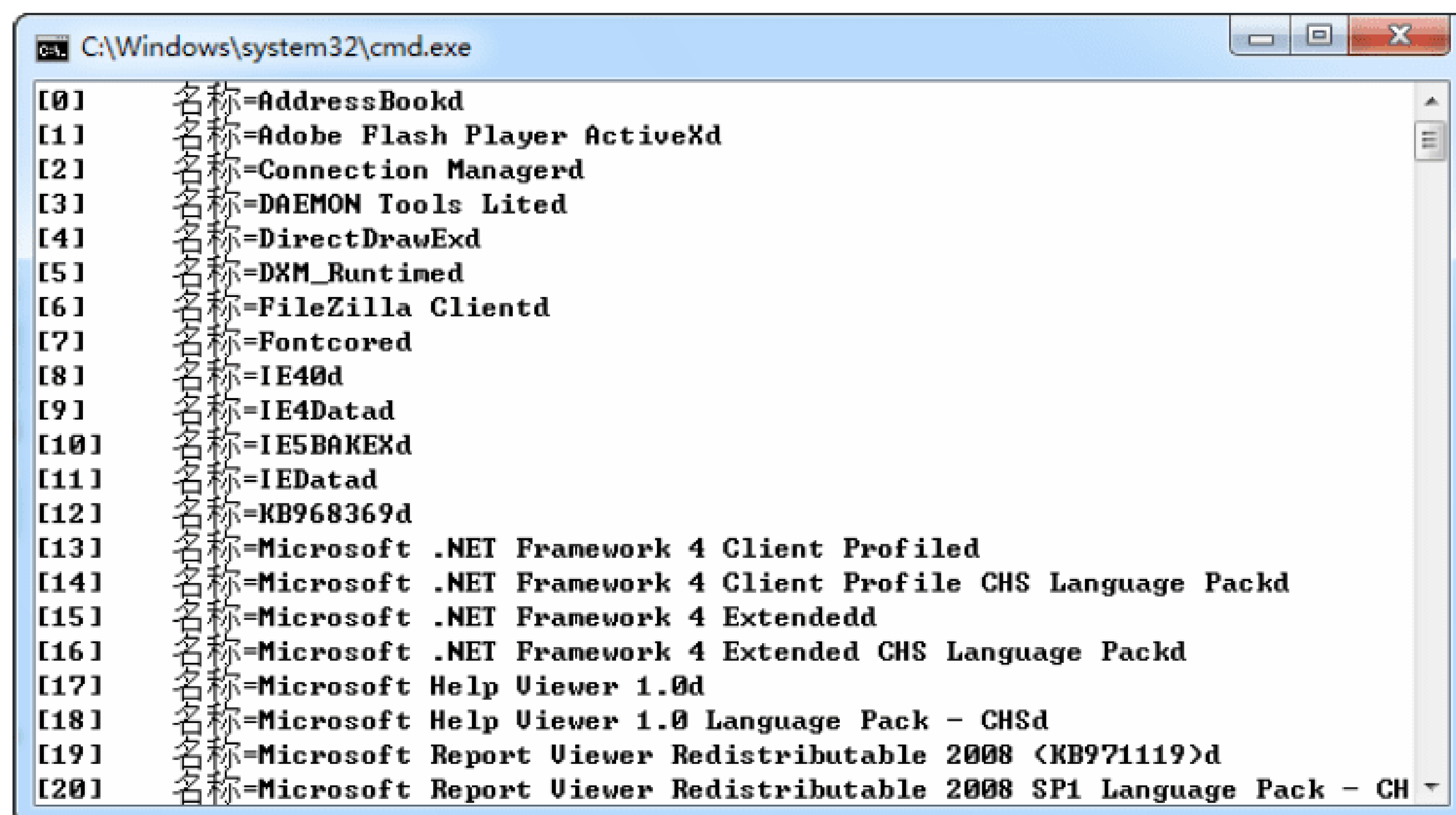


图 21-27　枚举安装程序运行效果

21.5　INI 文件的读写函数

在 Win32 程序中，一般将初始化信息存储在扩展名为 INI 的文件中，并且提供了读写 INI 文件的函数。使用这些函数不仅可以读写字符串数据和整型数据，还可以读写结构数据，并且可以查询指定节名或键值的数据。本节将介绍有关 INI 文件的读写函数。在 IniSample 示例中可以找到有关本节的示例代码。

21.5.1　向指定键写入字符串

INI 文件适合存放少量的配置性数据。可以实现分节存储，节以中括号括起来。每节下面可以存放键名和数据对，用于定义指定参数的值，在键名和数据之间以等号分隔。代码如下：

```
[section]
key=string
```

其中，section 指定节名，key 指定键值，string 指定键值对应的取值。如系统配置文件 config.ini，可以分为数据库参数节和通信配置节。在通信配置节下会有 IP 键名，表示要连接的 IP 地址。此时 config.ini 文件如下：

```
[Database]
DB=Test
```

```
User=sa
[Comm]
IP=127.0.0.1
```

在 Win32 中，使用 WritePrivateProfileString()函数可以向 INI 文件中的指定键值中写入字符串数据。其函数原型为：

```
BOOL WritePrivateProfileString(
  LPCTSTR lpAppName,          //指定要写入字符串数据所在的节的名称
  LPCTSTR lpKeyName,          //指定要写入的字符串对应的键值
  LPCTSTR lpString,           //指定要写入的字符串
  LPCTSTR lpFileName );       //指定要写入字符串数据的 INI 文件名
```

其中，如果 lpKeyName 参数为 NULL，则整节包括其中所有的条目都会被删除。如果 lpFileName 参数给出的不是文件的完整路径，则函数会查找 Windows 目录，如果文件不存在，则函数会创建文件。

如果函数操作成功，则返回非 0（true）；否则返回 0（false）。要获取错误原因，可以调用 GetLastError()函数。下面代码显示了此函数的用法。

```
01  bool WriteIniKeyString()    //向 INI 文件中指定的键值下写入字符串数据
02  {
03      if (!WritePrivateProfileString ("MaMa", "First",
04          "Sleep", "hobby.ini"))
05          return false;
06      if (!WritePrivateProfileString ("MaMa", "Second",
07          "Music", "hobby.ini"))
08          return false;
09      if (!WritePrivateProfileString ("Baby", "First",
10          "Milk", "hobby.ini"))
11          return false;
12      if (!WritePrivateProfileString ("Baby", "Second",
13          "Dance", "hobby.ini"))
14          return false;
15      if (!WritePrivateProfileString ("Baby", "Number",
16          "5", "hobby.ini"))
17          return false;
18      return true;
19  }
20  void RunWriteIniKeyString()
21  {
22      if (!WriteIniKeyString())
23          printf("遗憾——向 INI 文件中指定的键值下
24          写入字符串数据失败!\n");
25      else
26          printf("恭喜——向 INI 文件中指定的键值下
27      写入字符串数据成功!\n");
28  }
```

上面代码在 WriteIniKeyString()函数中调用 Win32 API 函数 WritePrivateProfileString()，将数据写入 hobby.ini 文件中，分别记录妈妈和宝宝的爱好。在 RunWriteIniKeyString()函数中调用此函数。执行程序会将数据写入 Windows 目录下的 hobby.ini 文件中，如果此文件不存在，则会创建此文件。程序运行后，hobby.ini 文件的内容如下：

```
[MaMa]
First=Sleep
Second=Music
[Baby]
```

```
First=Milk
Second=Dance
Number=5
```

21.5.2　获取指定键下的整型数据

Win32 中，可以使用 GetPrivateProfileInt()函数获取 INI 文件中指定键值下的整型数据。此函数会查找指定 INI 文件中的指定节下指定键值的数据，如果查找到了此键值，则函数会返回查找到的值，如果没有查找到此键值，则函数会返回传入的默认值。其函数原型为：

```
UINT GetPrivateProfileInt(
  LPCTSTR lpAppName,          //指定包含指定键的节的名称
  LPCTSTR lpKeyName,          //指定要获取的整型数据对应的键值
  INT nDefault,               //指定当指定节下的指定键值不存在时，返回的默认数据
  LPCTSTR lpFileName );       //指定 INI 文件名
```

上面的函数返回值为查找到的指定节下的指定键值的整数值。如果指定的键不存在，则返回函数传入的默认值。如果返回的结果值小于 0，则函数的返回值为 0。以下代码是此函数的使用方法。

```
01  void GetIniKeyInt()      //获取 INI 文件中指定键值下的整型数据
02  {
03      UINT uiResult = GetPrivateProfileInt("Baby", "Number",
04          99,"hobby.ini" );
05      if (uiResult > 0 )
06      {
07          printf("恭喜—从 INI 文件中读取指定键值的整型数据成功! \n
08              Baby’s Number hobby is:%d", uiResult);
09      }
10      else
11      {
12          printf("遗憾—从 INI 文件中读取指定键值的整型数据失败! \n");
13      }
14  }
```

上面的代码调用 GetPrivateProfileInt()函数返回 hobby.ini 文件 Baby 节中的键值为 Number 的整型数据，如果没有此键值，则返回 99。函数的运行结果如图 21-28 所示。

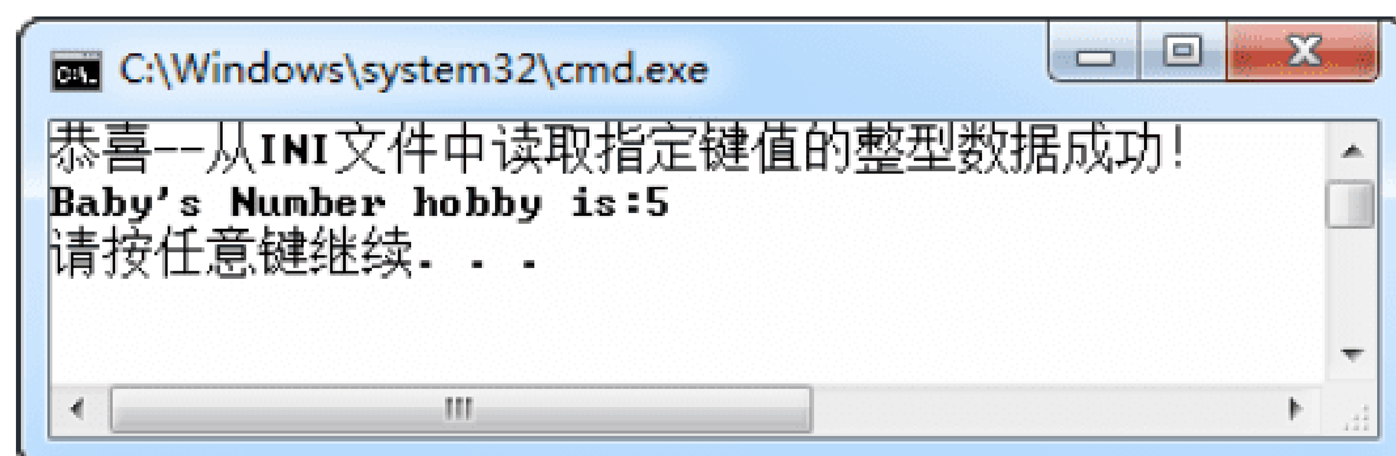


图 21-28　读取 INI 文件中指定键值的整型数据

21.5.3　获取指定键下的字符串数据

Win32 中，使用 GetPrivateProfileString()函数获取 INI 文件中指定键值下的字符串数据。此函数会查找指定 INI 文件中的指定节下指定键值的数据。如果查找到了此键值，则函数

会将其值存入指定缓冲区中，如果没有查找到此键值，则会将函数传入的默认值存入指定缓冲区中，并返回结果缓冲区中的字符数。其函数原型为：

```
DWORD GetPrivateProfileString(
  LPCTSTR lpAppName,         //指定包含指定键的节的名称
  LPCTSTR lpKeyName,         //指定要获取的字符串数据对应的键值
  LPCTSTR lpDefault, //指定当指定节下的指定键值不存在时，返回到缓冲区中的默认数据
  LPTSTR lpReturnedString,   //存放获取的字符串的缓冲区的指针
  DWORD nSize,               //指定结果缓冲区大小
  LPCTSTR lpFileName );      //指定 INI 文件名
```

函数参数 lpReturnedString 指向缓冲区。如果 lpAppName 参数和 lpKeyName 参数都不为 NULL，但是 lpReturnedString 中存放不下返回的数据，则会用 NULL 字符截断结果值，并且返回值为 nSize–1；如果 lpAppName 参数或 lpKeyName 参数都为 NULL，并且 lpReturnedString 中存放不下返回的数据，则会用两个 NULL 字符截断结果值，并且返回值为 nSize–2。以下代码是此函数的使用方法。

```
01  void GetIniKeyString()          //获取 INI 文件中指定键值下的字符串数据
02  {
03      char sResult[256];          //结果变量
04      DWORD dwResult, iSize = 256;//结果变量
05      dwResult = GetPrivateProfileString("Baby", "First", "未知",
06          sResult,iSize, "hobby.ini" );
07      if (dwResult > 0 )
08      {
09          printf("恭喜—从 INI 文件中读取指定键值的字符串数据成功! \n
10              Baby's First hobby   is:%s", sResult);
11      }
12      else
13      {
14          printf("遗憾—从 INI 文件中读取指定键值的字符串数据失败! \n");
15      }
16  }
```

上面的代码调用 GetPrivateProfileString()函数返回 hobby.ini 文件 Baby 节中的键值为 First 的字符串的数据，如果没有此键值，则返回“未知”。函数的运行结果如图 21-29 所示。

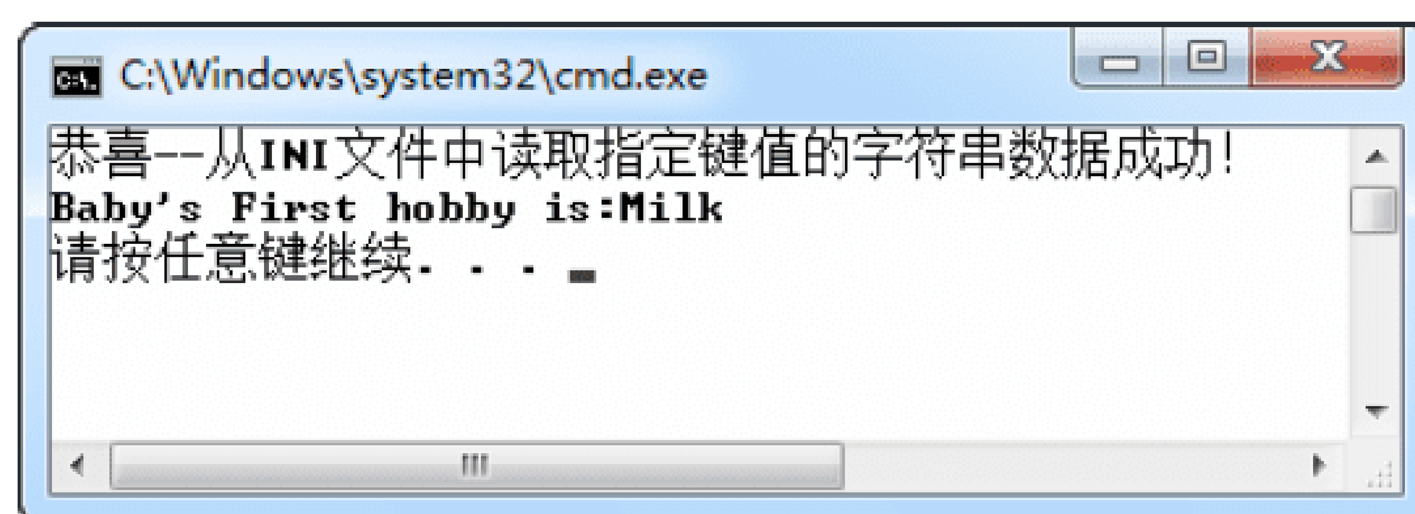


图 21-29　读取 INI 文件中指定键值的字符串数据

21.5.4　向 INI 文件写入结构数据

在 Win32 中，使用 WritePrivateProfileStruct()函数可以向 INI 文件中的指定键值中写入结构数据。在写入结构数据后，函数会计算校验和，并将其添加到数据的结尾处，使用校验和可以确保数据的完整性。其函数原型为：

```
BOOL WritePrivateProfileStruct(
  LPCTSTR lpszSection,  //指定要写入的结构数据所在的节的名称
  LPCTSTR lpszKey,      //指定要写入的字符串对应的键值
  LPVOID lpStruct,      //指定要写入的数据的指针
  UINT uSizeStruct,     //指定存放数据的结构的大小
  LPCTSTR szFile);      //指定要写入结构数据的 INI 文件名
```

如果函数操作成功，则返回非 0（true）；否则返回 0（false）。要获取错误原因，可以调用 GetLastError()函数。下面代码显示了此函数的用法。

```
01  struct student                //向 INI 文件中指定的键值下写入结构数据
02  {
03      int      ID;              //学生编号
04      int      age;             //学生年龄
05      char name[20];            //学生姓名
06  };
07  bool WriteIniKeyStruct()      //向 INI 文件中写结构
08  {
09      student st;               //定义 student 结构变量
10      st.ID = 1;                //分量赋值
11      st.age = 20;
12      memset(st.name, 0x00, sizeof(st.name));
13      strcpy(st.name, "张三");
14      //写入结构值
15      if (!WritePrivateProfileStruct ("Student", "First", &st,
16          sizeof(st),"student.ini"))
17          return false;
18      return true;
19  }
20  void RunWriteIniKeyStruct()//运行测试程序
21  {
22      if (!WriteIniKeyStruct())
23          printf("遗憾—向 INI 文件中指定的键值下写入结构数据失败!\n");
24      else
25          printf("恭喜—向 INI 文件中指定的键值下写入结构数据成功!\n");
26  }
```

在 WriteIniKeyStruct()函数中调用 Win32 API 函数 WritePrivateProfileStruct()，将数据写入 student.ini 文件中，记录 Student 结构的学生记录。在 RunWriteIniKeyStruct()函数中调用此函数。执行程序会将数据写入 Windows 目录下的 student.ini 文件中，如果此文件不存在，则会创建此文件。程序运行后，student.ini 文件的内容如下：

```
[Student]
First=0100000014000000D5C5C8FD00000000000000000000000000000074
```

从上面的结果中可以看出，在结构实际存储时，由位进行存储，并且在最后加入了校验码。

21.5.5　获取 INI 文件结构数据

Win32 中，使用 GetPrivateProfileStruct()函数获取 INI 文件中指定键值下的结构数据。此函数在获取结构后，会将获取的校验码与计算的校验码进行比对，进行数据完整性的验

证。此函数会查找指定 INI 文件中的指定节下指定键值的数据。如果查找到了此键值，则函数会将其值存入指定缓冲区中。如果没有查找到此键值，会将函数传入的默认值存入指定缓冲区中，并返回结果缓冲区中的字符数。其函数原型为：

```
BOOL GetPrivateProfileStruct(
  LPCTSTR lpszSection,   //指定包含指定键的节的名称
  LPCTSTR lpszKey,       //指定要获取的结构数据对应的键值
  LPVOID lpStruct,       //用于存放获取的结构数据的缓冲区的指针
  UINT uSizeStruct,      //指定结构缓冲区大小
  LPCTSTR szFile)        //指定 INI 文件名
```

如果函数操作成功，则返回非 0（true）；否则返回 0（false），要获取错误原因，可以调用 GetLastError()函数。下面代码显示了此函数的用法。

```
01  void GetIniKeyStruct()                            //获取结构数据
02  {
03      student st;                                   //定义结构变量
04      memset(st.name, 0x00, sizeof(st.name)); //初始化结构变量值
05      if (GetPrivateProfileStruct("Student", "First", &st,
06          sizeof(st),"student.ini" ))               //获取结构信息
07      {
08          printf("恭喜—从 INI 文件中读取指定键值的字符串数据成功! \n
09              学号=%d\n 姓名=%s\n年龄=%d", st.ID,st.name,st.age);
10      }
11      else
12      {
13          printf("遗憾—从 INI 文件中读取指定键值的字符串数据失败! \n");
14      }
15  }
```

上面的代码调用 GetPrivateProfileStruct()函数，返回 student.ini 文件中 Student 节中的键值为 First 的结构数据。函数的运行结果如图 21-30 所示。

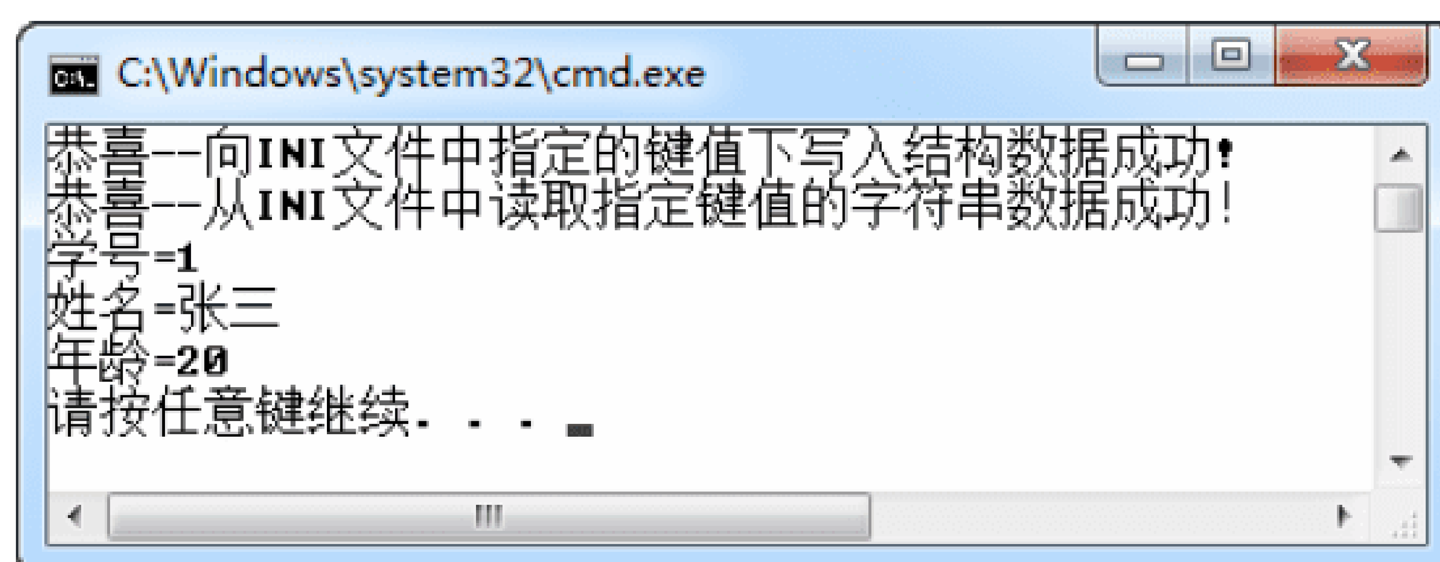


图 21-30　读取 INI 文件结构数据的效果

21.5.6　向指定节写入数据

在 Win32 中，使用 WritePrivateProfileSection()函数向 INI 文件中的指定节中写入数据，替换 INI 文件中指定节的键值及其取值。其函数原型为：

```
BOOL WritePrivateProfileSection(
  LPCTSTR lpAppName,               //指定要写入的数据所在的节名称
  LPCTSTR lpString,                //指定要写入的键名及其键值
  LPCTSTR lpFileName );            //指定要写入字符串数据的 INI 文件名
```

如果没有与参数指定的节名匹配的节，则函数会在 INI 文件尾添加新节，并将 lpString 指定的键名和键值对写入当前节下。如果查找到匹配的节，则函数会删除当前节下的内容，并将 lpString 指定的键名和键值对写入当前节下。lpString 参数中可以包含一个或多个字符串，最后以 NULL 结束。每个键名和键值对的格式如下：

```
key=string
```

如果函数操作成功，则返回非 0（true）；否则返回 0（false），要获取错误原因，可以调用 GetLastError()函数。下面代码显示了此函数的用法。

```
01  bool WriteIniSection()      //向 INI 文件中指定的节下写入数据
02  {
03      if (!WritePrivateProfileSection ("MaMa",
04          "Third=Read\r\nNumber=5","hobby.ini"))
05          return false;
06      if (!WritePrivateProfileSection ("Baby",
07          "Third=Toy\r\nForth=Play","hobby.ini"))
08          return false;
09      return true;
10  }
11  void RunWriteIniSection()   //运行测试函数
12  {
13      if (!WriteIniSection())
14          printf("遗憾—向 INI 文件中指定的节下写入数据失败!\n");
15      else
16          printf("恭喜—向 INI 文件中指定的节下写入数据成功!\n");
17  }
```

在 WriteIniSection()函数中调用 Win32 API 函数 WritePrivateProfileSection()，将数据写入 hobby.ini 文件中，分别记录妈妈和宝宝的爱好。在 RunWriteIniSection()函数中调用此函数。执行程序会将数据写入 Windows 目录下的 hobby.ini 文件中，如果此文件不存在，则会创建此文件。程序运行后，hobby.ini 文件的内容如下：

```
[MaMa]
Third=Read
Number=5
[Baby]
Third=Toy
Forth=Play
```

21.5.7　获取所有节名

Win32 中，使用 GetPrivateProfileSectionNames()函数获取 INI 文件中的所有节名。此函数会查找指定 INI 文件中的所有节名，并将结果存入缓冲区中。其函数原型为：

```
DWORD GetPrivateProfileSectionNames(
  LPTSTR lpszReturnBuffer,           //指定包含所有节名的缓冲区的指针
  DWORD nSize,                       //缓冲区的大小
  LPCTSTR lpFileName);               //指定 INI 文件名
```

其中，如果 lpFileName 参数为 NULL，则函数会从 WIN.INI 文件中获取所有节名。函数返回值为获取的存放节名的字符串缓冲区中的有效字符数。如果返回的字符数超过 nSize，则结果值会进行截取，并返回 nSize–2。以下代码是此函数的使用方法。

```
01  void GetSectionNames()                          //获取节名
02  {
03      char sResult[256];                          //定义返回值变量
04      DWORD dwResult, iSize = 256;
05      dwResult = GetPrivateProfileSectionNames(sResult,
06          iSize,"hobby.ini");                     //获取节名
07      if (dwResult > 0 )
08      {                                           //输出获取的结果
09          printf("恭喜—获取节名成功! 长度=%d\n", dwResult);
10          char item[256];
11          int  len=0;
12          int strEnd = 0x00;
13          memset(item, 0x00, sizeof(item));
14          for (int i = 0; i<dwResult ; i++)       //循环取出节名
15          {
16              if (sResult[i] != strEnd)
17              {
18                  item[len] = sResult[i];
19                  len++;
20              }
21              else
22              {
23                  printf("%s\n", item);
24                  memset(item, 0x00, sizeof(item));
25                  len = 0;
26                  if (sResult[i+1] == strEnd)
27                  {
28                      break;
29                  }
30              }
31          }
32      }
33      else                                        //输出错误结果
34      {
35          printf("遗憾—获取节名失败! \n");
36      }
37  }
```

上面的代码调用 GetPrivateProfileSectionNames()函数，返回 hobby.ini 文件中所有的节名。在此示例中，要注意结果字符串的处理。此处是遍历结果缓冲区，当遇到 NULL 字符时，则将当前项显示出来，继续处理字符，直到连续两个 NULL 字符或处理的长度超过结果缓冲区的长度。函数的运行结果如图 21-31 所示。

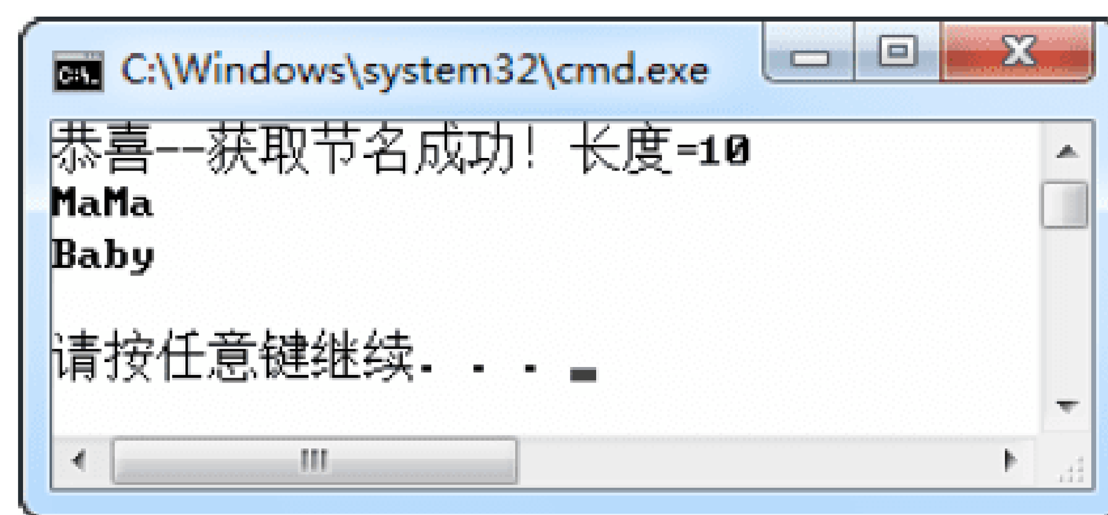


图 21-31　读取 INI 文件中的所有节名效果

21.5.8　获取指定节的键名及数据

Win32 中，使用 GetPrivateProfileSection()函数获取 INI 文件中指定节下的键名和数据，并将结果存入缓冲区中。其函数原型为：

```
DWORD GetPrivateProfileSection(
  LPCTSTR lpAppName,              //指定要获取的数据所在的节名
  LPTSTR lpReturnedString,        //指定包含键名和数据的缓冲区的指针
  DWORD nSize,                    //缓冲区的大小
```

```
  LPCTSTR lpFileName);            //指定 INI 文件名
```

函数返回值为获取的存放键名和数据的字符串缓冲区中的有效字符数。如果返回的字符数超过 nSize，则结果值会进行截取，并返回 nSize–2。返回的数据中，每对键名和数据的格式如下：

```
key=string
```

其中，key 表示键名，string 表示对应键名的数据。以下代码是此函数的使用方法。

```
01  void GetSectionKeyAndData()                //获取指定节下的键名和数据
02  {
03      char sResult[256];                     //变量定义
04      DWORD dwResult, iSize = 256;
05      dwResult = GetPrivateProfileSection("Baby", sResult, iSize,
06          "hobby.ini" );                     //获取值对
07      if (dwResult > 0 )                     //输出提示信息
08      {
09          printf("恭喜—获取 Baby 节下的键名和数据成功! 长度=%d\n", dwResult);
10          char item[256];
11          int  len=0;
12          int strEnd = 0x00;
13          memset(item, 0x00, sizeof(item));
14          //使用 for 循环，遍历出所有的数据
15          for (int i = 0; i<dwResult ; i++)
16          {
17              if (sResult[i] != strEnd)
18              {
19                  item[len] = sResult[i];
20                  len++;
21              }
22              else                           //输出值对
23              {
24                  char* pdest;
25                  char key[256], data[256];
26                  pdest = strchr(item, '=');
27                  memset(key, 0x00, sizeof(key));
28                  memcpy(key, item, pdest-item);
29                  memset(data, 0x00, sizeof(data));
30                  strcpy(data, pdest+1);
31                  printf("键名=%s        数据=%s\n", key, data);
32                  memset(item, 0x00, sizeof(item));
33                  len = 0;
34                  if (sResult[i+1] == strEnd)   break;
35              }
36          }
37      }
38      else
39          printf("遗憾—获取 Baby 节下的键名和数据失败! \n");
40  }
```

上面的代码调用 GetPrivateProfileSection()函数，返回 hobby.ini 文件中 Baby 节中的键名和数据对。接着是遍历结果缓冲区，当遇到 NULL 字符时，处理当前项，分别显示出键名和数据。然后继续处理字符，直到连续两个 NULL 字符或处理的长度超过结果缓冲区的长度。函数的运行结果如图 21-32 所示。

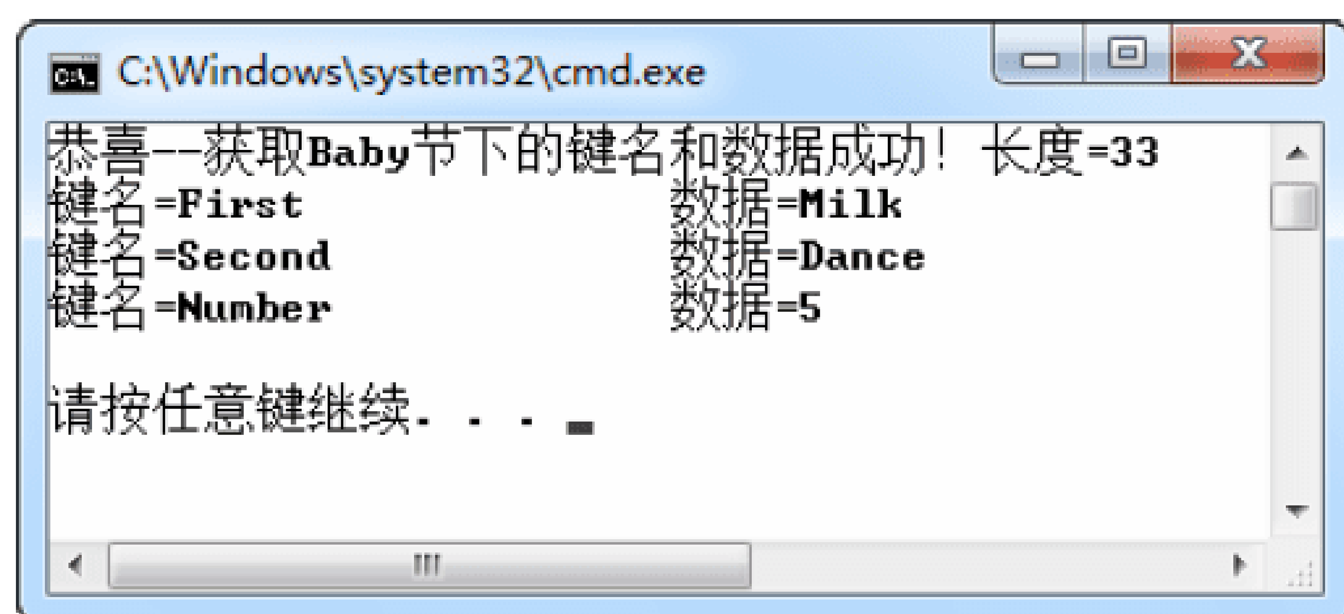


图 21-32　读取 INI 文件中指定节名下的键名及数据的程序效果

21.6　XML 文件操作

XML（eXtensible Markup Language，可扩展标记语言）是目前事实上的数据传输工业标准，是基于内容来组织文件格式的。随着互联网的飞速发展，要求使用跨平台的、统一的文件格式支持互联，因此出现了 XML 技术。本节将简单地介绍有关 XML 文件的操作。

21.6.1　XML 文件简介

XML 标准是 SGML（Standard Generalized Markup Language，标准通用标记语言）的一个子集，由 SGML 工作组在 1996 年建立。XML 文件的核心就是“有头有尾”，所有数据都使用数据头和数据尾将其包括起来，而数据头和数据尾采用数据标记实现，作用一是确定数据的范围，二是通过数据标记标识数据种类。下面是一个典型的 XML 文件。

```
<Student>
    <ID>200801</ID>
    <NAME>张三</NAME>
    <SEX>男</SEX>
</Student>
```

上面举了一个简单的 XML 例子，所有复杂的 XML 文件都是从这种基本格式衍生而来的。标记<Student>表示一条学生信息，其中包含 3 项信息：ID、NAME 和 SEX，这 3 项信息分别以<ID>、<NAME>和<SEX>标记标识。因此可以看出，XML 文件比其他文件格式更易于扩展和理解。

21.6.2　XML 文件的优势

XML 文件作为一种跨平台的数据交换格式，有其特有的优势，促使其发展到今天。

- ❑ 因为 XML 符合 SGML 规范，因此 XML 数据可以在万维网 WWW 上传输，但是不一定是 HTML。确切地说，HTML 只是 XML 标准的一个子集。XML 标准分为很多子集，如有关数学方面的 XML 标准 MathML、可缩放矢量图形的 XML 标准 SVG、有关化学方面的 XML 标准 CML 等。虽然这些专业领域的 XML 标准的定义不同，但是传输方式都是一致的。
- ❑ XML 通过 DTD 文件或 XML 数据架构支持“自解析”文档，即 XML 文档可以根据内容实现自我解释。XML 文件在开头部分带有描述文档中信息类型的指令。可

以根据这些指令判断 XML 文档中包含的资源类型及解析方式。

- 为了支持应用多样化，XML 允许用户自定义文档标记，使用这些文档标记，用户可以描述业务数据。在处理这些自定义标记时，XML 解析器可以通过读取相应的 DTD 文件实现。

基于 XML 文件的种种优势和目前业界对于 XML 标准的支持，在编写互联软件时，需要掌握操作 XML 文件的编程知识。下面两小节将简单地介绍如何读写 XML 文件的内容。

21.6.3　读取 XML 文件内容

由于 XML 文件是符合一定标准的可以扩展的标记语言，所以其内容标签不是固定的，因此也就不能以固定的内容标签解析 XML 文件，而是应该根据 XML 标准的格式编写出符合规范的可以解析所有符合 XML 标准的 XML 文件。幸好，业界目前对 XML 的支持是很广的。因为本书是介绍 Visual Studio 2010 环境下的开发，因此，这里以微软公司的 MSXML 解析器为例，讲解如何操作 XML 文件。

MSXML 解析器在处理 XML 文件时，将其作为“树”的数据结构处理。XML 文件的根元素作为“根结点”，其余依次为“树枝结点”和“树叶结点”。

本小节以一个读取学生记录 XML 文件为例，说明如何读取 XML 文件。代码如下：

```
01  void CXMLParserSampleDlg::OnButtonReadXml()//读取 XML 文件
02  {
03      Cstring log, info;                          //定义变量
04      MSXML2::IXMLDOMDocumentPtr pDoc;            //创建 DOMDocument 对象
05      if (!SUCCEEDED(pDoc.CreateInstance
06          (__uuidof(MSXML2::DOMDocument30))))
07          WriteLog("创建 DOMDocument 对象失败，请确认安装 MSXMLParser 组件!");
08      pDoc->load("Students.xml");                 //加载文件
09      MSXML2::IXMLDOMElementPtr  pChild;          //定义 XML 结点元素
10      pChild=(MSXML2::IXMLDOMElementPtr)
11          (pDoc->selectSingleNode("//Students"));
12      //查询结点
13      BSTR var;                                   //定义变量
14      VARIANT varVal;                             //定义变量
15      pChild->get nodeName(&var);                 //结点名称和结点值
16      pChild->get_nodeTypedValue(&varVal);        //获取结点值
17      info.Format("结点名称=%s\t 值=%s\r\n",
18          (char*)(_bstr_t)pChild->nodeName,
19          (char*)(_bstr_t)pChild->nodeTypedValue);//输出提示信息
20      log += info;                                //记录信息
21      MSXML2::IXMLDOMNodeListPtr pList;           //结点链表变量
22      MSXML2::IXMLDOMNodePtr pNode;               //结点
23      pList = (MSXML2::IXMLDOMNodeListPtr)
24          pChild->selectNodes("Student");
25      //查询结点
26      long nodeCount = pList->Getlength();        //获取结点个数
27      for (int i = 0;i <nodeCount;i++)            //使用 for 循环依次检索结点
28      {
29          if (!SUCCEEDED(pList->get item(i, &pNode)))
30              continue;
31          BSTR var;
```

```
32          VARIANT varVal;
33          pNode->get_nodeName(&var);            //结点名称和结点值
34          pNode->get_nodeTypedValue(&varVal);
35          info.Format("\r\n 结点名称=%s\t 值=%s\r\n",
36              (char*)(_bstr_t)pNode->nodeName,
37              (char*)(_bstr_t)pNode->nodeTypedValue);//格式化提示信息
38          log += info;
39          MSXML2::IXMLDOMNamedNodeMapPtr pAttrs=NULL;
40          MSXML2::IXMLDOMNodePtr   pAttrItem;
41          pNode->get_attributes(&pAttrs);
42          long   nCount;
43          pAttrs->get_length(&nCount);
44          for(int j=0; j<nCount; j++)           //输出属性值
45          {
46              if (!SUCCEEDED(pAttrs->get_item(j,&pAttrItem)))
47                  continue;
48              info.Format("属性名称=%s\t 值=%s\r\n",
49                  (char*)(_bstr_t)pAttrItem->baseName,
50                  (char*)(_bstr_t)pAttrItem->text);
51              log += info;
52          }
53      }
54      WriteLog(log);                            //显示日志信息
55  }
```

上面代码首先创建 DOMDocument 对象用于表示一个 XML 文件，调用 load()函数装载要读取的 XML 文件。装载后，读取 Students 结点的信息并显示出来，然后读取 Students 结点的所有子结点，显示出子结点的名称和值后，显示出结点的所有属性。程序运行效果如图 21-33 所示。

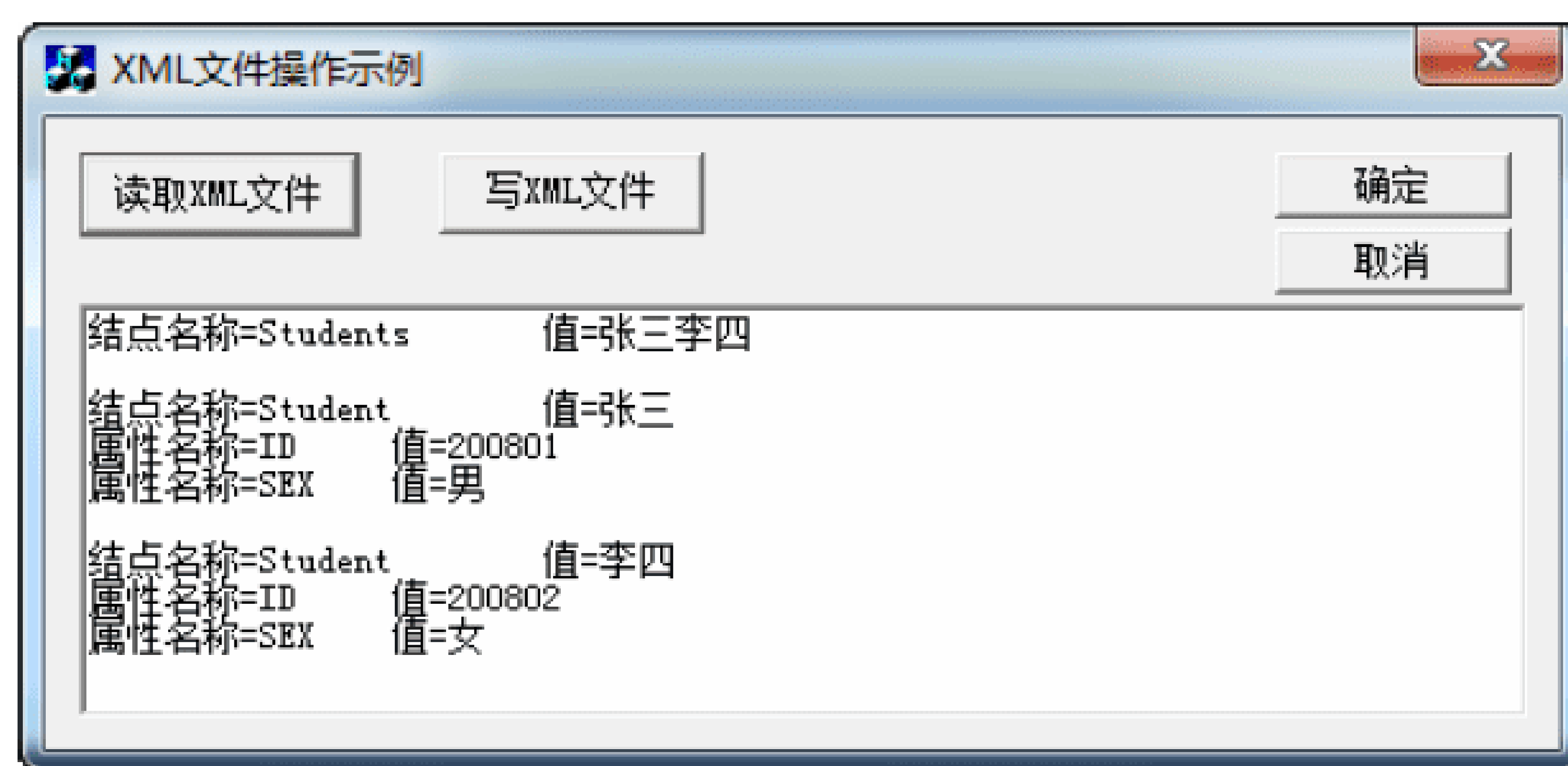


图 21-33　读 XML 文件内容运行效果

21.6.4　向 XML 文件中写入内容

向 XML 文件写内容是从 XML 文件中读内容的逆过程。本小节以简单地写入两条学生记录为例，说明如何向 XML 文件中写内容。代码如下：

```
01  void CXMLParserSampleDlg::OnButtonWriteXml()    //向 XML 文件中写入内容
02  {
03      MSXML2::IXMLDOMDocumentPtr pDoc;               //文档对象
04      MSXML2::IXMLDOMElementPtr  pRoot;              //根结点对象
05      if (!SUCCEEDED(pDoc.CreateInstance
06          (__uuidof(MSXML2::DOMDocument30))))
```

```
07            WriteLog("创建 DOMDocument 对象失败，请确认安装 MSXMLParser 组件!");
08        pDoc->raw_createElement((_bstr_t)(char*)"Students",
09            &pRoot);
10        //根结点的名称为 Student
11        pDoc->raw appendChild(pRoot, NULL);          //增加结点
12        MSXML2::IXMLDOMElementPtr  pChild;
13        pDoc->raw_createElement((_bstr_t)(char*)("Student"),
14            &pChild);
15        pChild->Puttext("张三");//节点值
16        pChild->setAttribute("ID","200801");         //属性名，属性值
17        pChild->setAttribute("SEX","男");            //设置属性值
18        pRoot|->ppendChild(pChild);                  //添加结点
19        pDoc->raw_createElement((_bstr_t)(char*)("Student"),
20            &pChild);
21        //创建元素
22        pChild->Puttext("李四");                     //赋值
23        pChild->setAttribute("ID","200802");         //设置 ID 属性值
24        pChild->setAttribute("SEX","女");            //设置 SEX 属性值
25        pRoot->appendChild(pChild);                  //增加结点值
26        pDoc->save("Students.xml");                  //保存到文件
27        WriteLog("存入 XML 文件成功");               //输出提示信息
28    }
```

上面代码在创建 DOMDocument 对象后，首先创建根结点 Students，然后向根结点添加两条具有属性值的 Student 结点，最后调用 save()函数将 XML 内容保存到 XML 文件中。程序运行效果如图 21-34 所示，是生成的 Students.xml 文件的内容。

```
- <Students>
    <Student ID="200801" SEX="男">张三</Student>
    <Student ID="200802" SEX="女">李四</Student>
  </Students>
```

图 21-34　写 XML 文件后的内容

21.7　本 章 小 结

本章介绍了 3 种 Windows 系统中常用的数据存储文件格式——注册表、INI 文件和 XML 文件。本章重点介绍了注册表的读写方法、INI 文件的读写方法和 XML 文件操作。本章的难点是在这 3 种文件读写方法中选择合适的方法。第 22 章将介绍实现模块化技术的动态链接库的编程方法。

21.8　习　　题

1．编程完成下列操作。

（1）创建注册表项：HKEY_LOCAL_MACHINE\SOFTWARE\Test\MyTest。

（2）指定由（1）创建的注册表项的默认值。

（3）删除由（1）创建的注册表项。

【思路】（1）、（2）和（3）可以分别参考 21.1.3 小节、21.1.8 小节和 21.1.6 小节的示例。

2．参考 21.3.3 小节所讲的示例，使用 CRegKey 编程查询由第 1 题创建的注册表项的键值（需要保留由第 1 题创建的注册表项，即不执行第 1 题的（3）操作）。

【思路】使用 CregKey 类的 QueryValue()函数可以查询注册表键值。

3．在工程的目录下创建 INI 文件 test.ini，并添入以下内容：

```
[files]
filename=INI 文件测试
[Mail]
address=123456789@mymail.com
MAPI=1
```

完成以下操作。

（1）获取 filename 的值。

（2）创建新的节[Time]，节下添加键 record，值为 20120402。

【思路】（1）和（2）可以分别参考 21.5.3 小节和 21.5.6 小节的示例。

第 22 章　动态链接库编程

在应用程序开发的过程中，有的功能是在很多地方都可以重复使用，如有关数据的访问、协议转换等。为了提高这些功能的复用，减少系统开发工作量，系统提供了动态链接库（Dynamic-link libraries，DLL）技术，实现功能模块化。本章将讲述有关动态链接库的编程。

22.1　基 本 概 念

动态链接库是包含函数和数据的模块，将实现一定功能的函数和数据按照一定的规则封装在一起。本节将介绍有关动态链接库的基本概念及工作方式。

22.1.1　动态链接库的概念

简单地讲，DLL 就是完成一定功能的模块，既可以包含数据和函数，也可以包含类。DLL 最典型的例子——微软的 Win32 应用程序接口，就是通过一组动态链接库的方式实现的。因此，任何程序都可以通过调用动态库的方式使用 Win32 API，从而可以访问系统底层接口。DLL 中包含两种对象。

- 导出对象：如导出数据、导出函数和导出类，此种对象可以被其他可执行模块调用。虽然 DLL 可以导出数据，但是通常 DLL 中的数据都是内部数据，仅供内部函数使用，不建议从 DLL 中导出数据。
- 内部对象：如内部数据、内部函数和内部类，此种对象只能在 DLL 内部由内部使用。

DLL 动态链接库和 EXE 可执行文件类似，都是可执行程序模块，但是也存在很多不同之处。对于用户来说，最大的差别在于 DLL 不是可以直接执行的程序。从系统的角度来看，它们之间存在两个基本的区别，一是应用程序可以在系统中同步运行多个实例，而 DLL 只能有一个实例；二是应用程序可以管理诸如堆栈、全局内存、文件句柄和消息队列等资源，而 DLL 不能管理这些资源。DLL 的工作过程如图 22-1 所示。

在图 22-1 中列出了 DLL 被调用的工作过程。假定有 3 个 DLL，分别是 A.DLL、B.DLL 和 C.DLL，有两个程序，分别是程序 1 和程序 2。程序 1 需要调用 A.DLL 和 B.DLL，而程序 2 需要调用 B.DLL 和 C.DLL。

从图 22-1 可以看出，DLL 由调用模块在运行时进行装载，调用模块装载 DLL 后，将 DLL 映射到虚拟地址空间中，这样可以减少同一时间多个应用程序使用相同功能所重复使用的内存。因为虽然每个应用程序有自己的数据备份，但是共享相同的代码。这也是动态链接与静态链接的区别，对于静态链接，链接器需要复制函数代码到调用模块的数据空间中。

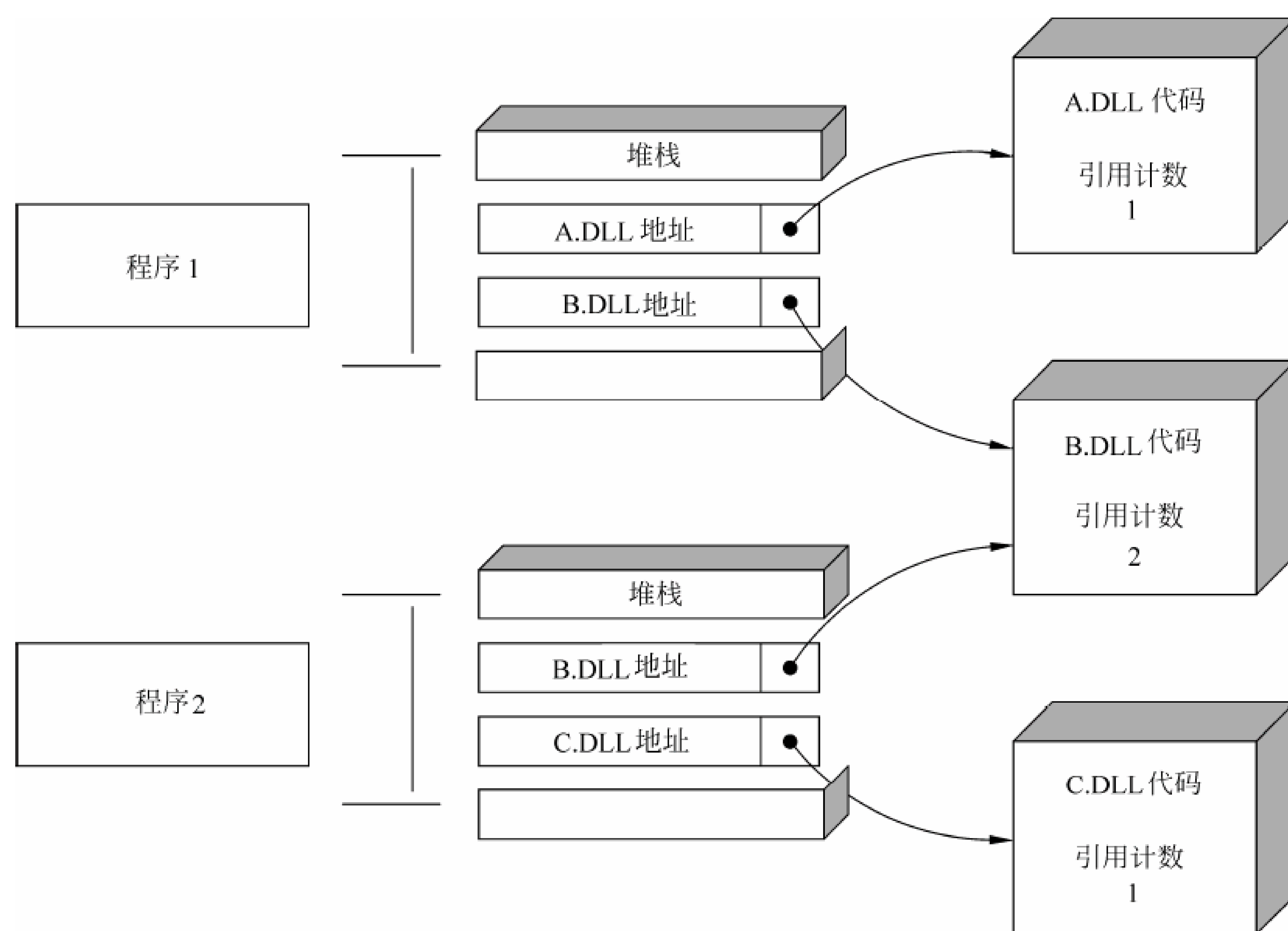


图 22-1　DLL 工作过程

同时，动态链接允许模块在装载时，仅包含系统需要的信息，或者运行时定位到导出 DLL 函数的代码。

对于被调用的 DLL 来说，系统会维护被引用的次数，每有一个进程调用 DLL，则 DLL 的引用计数会增加 1，每有一个引用 DLL 的进程终止，DLL 的引用计数会减 1。直到 DLL 的引用计数变成 0，DLL 会被系统卸载，退出地址空间。

但是需要注意的是，虽然调用 DLL 的进程都使用相同的 DLL 副本，但是调用 DLL 的进程所使用的 DLL 导出的函数是运行在调用进程或线程的上下文的，因此 DLL 在处理资源时，是这样处理的：

- 调用 DLL 的进程的线程可以使用 DLL 函数打开的句柄。同样，DLL 函数也可以使用调用 DLL 进程的线程打开的句柄。
- DLL 可以使用调用线程的堆栈和进程的虚拟地址空间。
- DLL 从调用进程的虚拟地址空间中分配内存。

总之，调用 DLL 是多个模块共享相同的 DLL 代码，但是每个模块使用的 DLL 又是工作在模块自己的上下文环境中的。

22.1.2　动态链接库的优点

因为动态链接库是将功能封装在一起的模块，因此，与将代码直接写入调用模块中相比，它不仅可以提高程序的复用，减少代码开发工作量，同时使得功能更新更方便。除了这些模块化带来的优点外，动态链接库的工作方式也决定了它先天具有比静态链接更多的优点，如下所述。

- 节约内存和减少交换：当应用程序使用动态链接时，多个进程可以同步使用一个 DLL，共享内存中 DLL 的单个副本。相比之下，当应用程序使用静态链接库时，Windows 必须为每个应用程序装载一个库代码的副本到内存中。

- 节约磁盘空间：当应用程序使用动态链接时，多个应用程序可以共享磁盘上的单个 DLL 副本。相比之下，当应用程序使用静态链接库时，每个应用程序要将库代码作为独立的副本链接到可执行镜像中。
- 当 DLL 中的函数修改时，只要函数参数、调用规定和返回值没有改变，使用 DLL 的应用程序不需要重新编译或链接。而静态链接的函数改变时，需要应用程序重新链接。
- 支持多语言编程：只要应用程序遵循相同的调用规范，则使用不同编程语言编写的程序可以调用相同的 DLL 函数。程序和 DLL 函数必须兼容——函数定义的参数入栈顺序、函数或应用程序谁来负责清理堆栈、参数是否传入寄存器中等方面必须兼容。
- 轻松地创建中间版本：通过将资源放入 DLL 中，使得创建应用程序的中间版本非常简单。如可以将应用程序的每个语言版本的字符串放到单独的一个资源 DLL 中，并为不同的语言版本装载合适的资源 DLL 就可以了。

虽然使用 DLL 有诸多的优点，但是也需要格外注意使用 DLL 的缺点。即调用 DLL 的应用程序不是独立的，程序的运行依赖于所使用的 DLL 是否存在。

22.1.3　DLL 的种类

使用 Visual Studio 2010，可以构建不使用 MFC 的 Win32 DLL 和使用 MFC 的 MFC DLL。非 MFC DLL 是内部不使用 MFC 类库的 DLL，在 DLL 中的导出函数既可以被 MFC 可执行文件调用，也可以被非 MFC 可执行文件调用。MFC DLL 分为 3 种开发方式。

- 使用静态链接 MFC 类库的常规 DLL。此种类型的 DLL 在内部使用 MFC，使用 MFC 的静态链接库版本构建。从其中导出的函数既可以被 MFC 可执行文件调用，也可以被非 MFC 可执行文件调用。通常从常规 DLL 中导出的函数使用标准 C 接口。
- 使用动态链接 MFC 类库的常规 DLL。此种类型的 DLL 在内部使用 MFC，使用 MFC 的动态链接库版本构建（也就是 MFC 的共享版本）。从此种 DLL 中导出的函数既可以被 MFC 可执行文件调用，也可以被非 MFC 可执行文件调用。
- MFC 扩展 DLL。此 DLL 通常完成从现有的 MFC 类库中的类派生而来的可以重复使用的类。扩展 DLL 使用 MFC 的动态链接库版本（也就是 MFC 的共享版本）构建。只有使用 MFC 共享版本的 MFC 可执行文件（应用程序或规则 DLL），才可以使用扩展 DLL。使用扩展 DLL，读者可以从 MFC 派生新的用户自定义类，并为调用 DLL 的应用程序提供 MFC 的扩展版本。扩展 DLL 也可以用于在应用程序和 DLL 之间传递 MFC 派生对象。

如果 DLL 没有使用 MFC，则可以使用 Visual Studio 2010 构建非 MFC WIN32 DLL。链接 DLL 到 MFC，不管是静态的还是动态的，都会明显占用磁盘空间和内存。除非 DLL 真正使用了 MFC，否则不要将 DLL 链接到 MFC。

如果 DLL 需要使用 MFC，并且既会被 MFC 应用程序使用，也会被非 MFC 应用程序使用，则必须构建动态链接到 MFC 的常规 DLL，或静态链接到 MFC 的常规 DLL。大多数情况下，使用动态链接 MFC 的常规 DLL 要比使用静态链接 MFC 的常规 DLL 更好，因为动态链接比静态链接时 DLL 的文件要小，并且使用 MFC 的共享版本会节约内存，使其

更有效。如果静态链接 MFC，DLL 的文件大小会更大，并且会潜在地占用额外的内存，因为需要装载其私有的 MFC 库代码副本。动态链接 MFC 的 DLL 要比静态链接 MFC 的 DLL 更快，因为前者不需要链接 MFC 本身。

使用动态链接 MFC 的缺点是，用户必须在 DLL 中分发共享的 DLL MFCx0.DLL 和 MSVCRT.DLL，或者是类似的文件。MFC DLL 是可以自由分发的，但是用户仍然需要在安装程序中安装这些 DLL。另外，程序必须加上 MSVCRT.DLL，因为它包含了应用程序和 MFC DLL 本身都用到的 C 运行时库。

如果 DLL 仅被 MFC 可执行文件使用，则在常规 DLL 或扩展 DLL 之间进行选择。如果 DLL 实现从现有的 MFC 类中派生而来的可重用类，或需要在应用程序和 DLL 之间传递 MFC 派生对象时，则必须使用扩展 DLL。如果 DLL 动态链接到 MFC，则 MFC DLL 必须与 DLL 一起发布。

22.1.4　DLL 文件的组成

DLL 文件与 EXE 文件相似，主要不同在于 DLL 文件包含一个导出表。此导出表中包含 DLL 导出给其他可执行文件的函数名称。这些函数是 DLL 的入口，并且只有在导出表中的函数才可以被其他可执行文件访问。DLL 中的其他函数是 DLL 私有的。

DEF 文件最少要包括下面的模块定义语句。

- 文件的第一条语句必须是 LIBRARY 语句。此语句定义了 DEF 文件所属的 DLL。LIBRARY 语句后写入 DLL 名称。链接器将此名称放入 DLL 的导入库。
- 文件中的 EXPORTS 语句用于列出导出的函数名称和为其分配序号值。格式是函数名称后写入@符号和序号值。函数的顺序可以任意分配，但是序号值的取值范围必须是 1～N，其中 N 是 DLL 导出的函数个数。
- 为了清晰，建议在 DLL 的 DEF 文件中使用 DESCRIPTION 语句描述 DLL 的功能，方便 DLL 的复用。
- 所有以分号开头的行都是注释行。

下面是 DEF 文件的例子。

```
; MFCDLL1.def : Declares the module parameters for the DLL.
LIBRARY      "MFCDLL1"
DESCRIPTION  'MFCDLL1 Windows Dynamic Link Library'
EXPORTS
    WriteLog    @1
```

22.2　DLL 的创建与使用实例

本节在 22.1 节的基础上，讲述 Win32 DLL 的创建方法和使用实例。列举了如何通过使用 DLL 获取其中的位图资源、替换程序中使用的对话框、屏蔽 Power 键和 Win 键，以及禁止使用<Alt+F4>键等。

22.2.1　创建 Win32 DLL

在 Visual Studio 2010 中创建 Win32 DLL 的步骤如下。

（1）选择“文件” | “新建”| “项目”命令，弹出“新建项目”对话框，选择“Win 32 项目”，如图 22-2 所示。

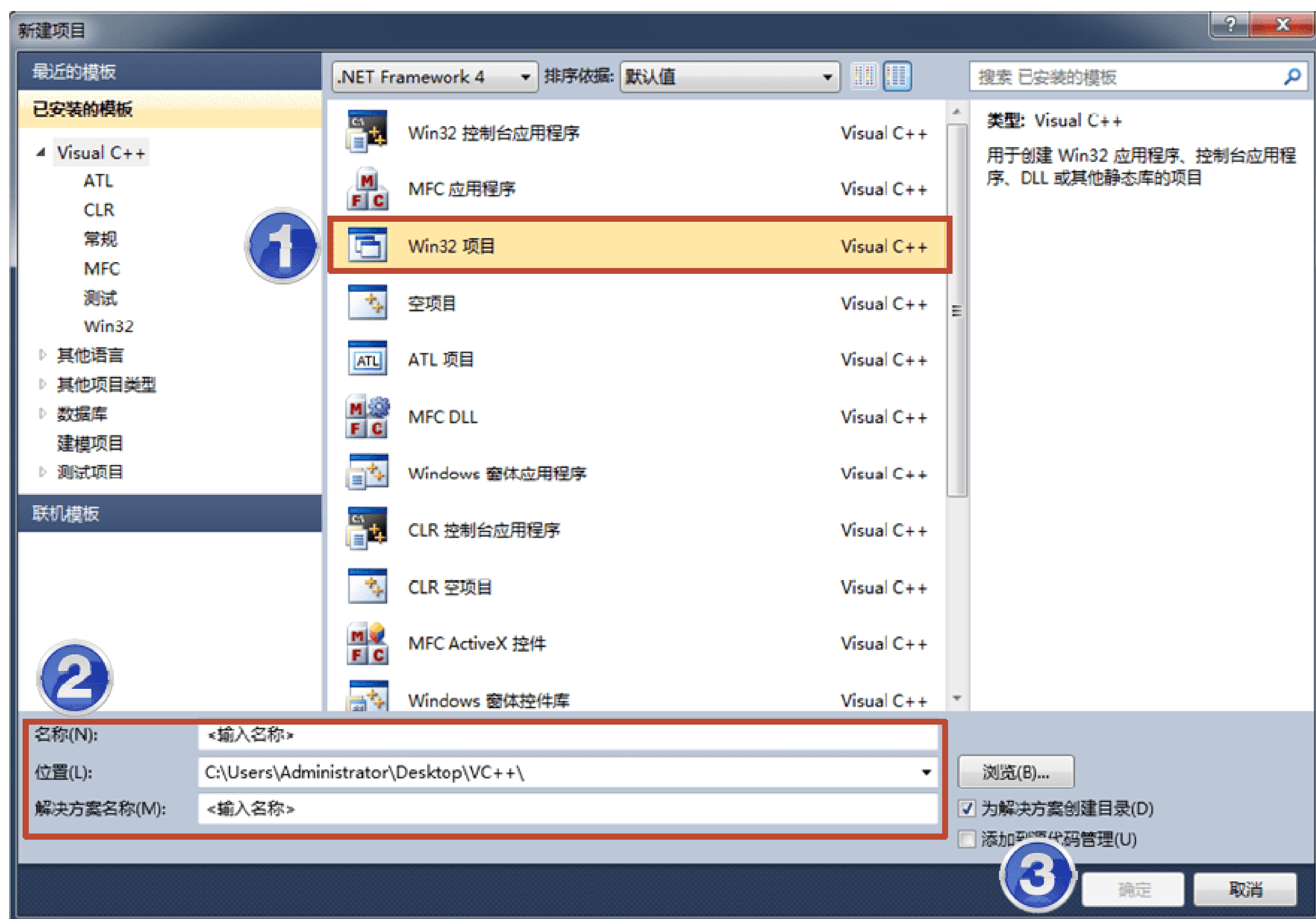


图 22-2　创建 Win32 DLL 的第一步

（2）在“名称”文本框和“位置”文本框中写入相应的值，单击“确定”按钮，弹出“Win32 应用程序向导”对话框，如图 22-3 所示。

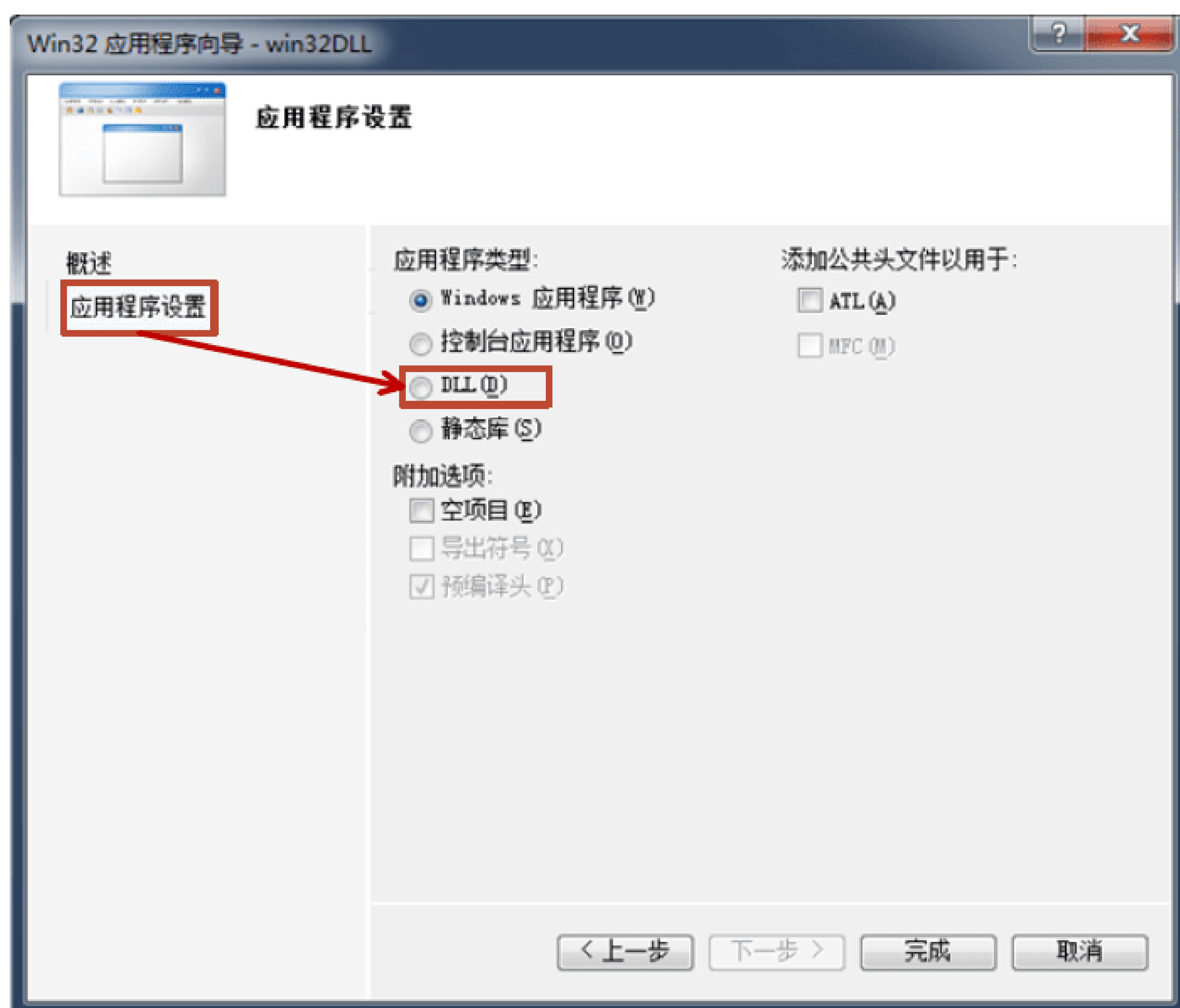


图 22-3　创建 Win32 DLL 的第二步

（3）在“应用程序类型”中选择“DLL”，再单击“完成”按钮，这样就成功地创建了一个 Win32 DLL。

在创建了 DLL 后，就需要将要执行的内容加入到 DLL 中。具体步骤如下。

（1）像前面讲的为工程添加文件一样，为 DLL 工程添加源代码文件。选择“项目”|“添加现有项”命令，在弹出的“添加现有项”对话框中选择要加入的源代码文件。

（2）加入函数名为 DllMain 的函数，并在此函数中为 DLL 增加初始化和终止代码。在本例中，因为选择了 A DLL that export some symbols 的 DLL 类型，系统自动增加了此函数的定义。

（3）确保使用__declspec(dllexport)关键字或 DEF 文件导出 DLL 的入口点。

（4）增加一个包含使用 DLL 函数定义的头文件。此头文件应该包含要用的函数的声明。当此头文件被 DLL 编译时，使用__declspec(dllexport)关键字从 DLL 中导出。当此头文件被使用 DLL 的应用程序编译时，则需要使用__declspec(dllimport)关键字从 DLL 中导入。

（5）如果 DLL 使用__declspec(dllexport)或 DEF 文件，系统会自动创建一个对应的导入库。当/IMPLIB 链接开关打开编译 DLL 时，应用程序需要导入库进行链接。

（6）构建 DLL。至此一个完整的 DLL 创建完成。

22.2.2　DLL 的导出

DLL 的导出表可以通过工具 DUMPBIN 使用/EXPORTS 开关查看。从 DLL 中导出函数有两种方法。

- 创建模块定义.DEF 文件（module-definition file），是包含一条或多条描述 DLL 不同属性的模块语句的文本文件。当构建 DLL 时，使用此 DEF 文件，可以按照函数的顺序号而不是名称从 DLL 中导出函数。
- 在函数定义中使用__declspec(dllexport)。如果不使用此关键字，则必须使用 DEF 文件。

如果 DLL 在多线程应用程序中使用，则需要 DLL 仅链接支持多线程的库，以使得 DLL 是“线程安全”。同样，需要确保访问全局数据的同步性，有关多线程的开发在第 23 章会详细介绍。

导入库（.LIB）文件包含链接器需要的导出 DLL 函数的外部引用需要的信息，因此，系统可以在运行时定位到指定的 DLL 和导出的 DLL 函数。如调用 CreateWindow()函数，用户必须在程序中链接导入库 USER32.LIB，因为 CreateWindow 在系统 DLL 中，而文件 USER32.DLL 用于解决调用 CreateWindow()函数的导入库。

每个 DLL 像应用程序一样必须有一个入口。当进程或线程装载或卸载 DLL 时，系统调用入口函数。如果程序像 C 运行库一样链接 DLL 到库中，则会提供一个入口函数，并允许提供一个独立的初始化函数。其中，DllMain 是一个用户自定义函数的占位符。当构建 DLL 时，用户必须指定使用的实际名称。系统在下面 4 种情况下调用入口函数。

- 进程装载 DLL 时。使用装载时动态链接的进程，DLL 在进程初始化时装载。对于使用运行时链接的进程，DLL 在 LoadLibrary()函数或 LoadLibraryEx()函数返回时装载。
- 进程卸载 DLL 时。当进程终止或调用 FreeLibrary()函数时，DLL 卸载，并且引用数目变成 0。如果进程是调用 TerminateProcess()函数或 TerminateThread()函数终止

的，系统不会调用 DLL 的入口。

- 已经装载 DLL 的进程创建新线程时。用户可以使用 DisableThreadLibraryCalls()函数关闭线程创建时的通知。
- 已经装载 DLL 的进程的线程正常终止时，但不是使用 TerminateThread()函数或 TerminateProcess()函数。但进程卸载 DLL 时，整个进程仅调用一次入口函数，而不是进程的每个存在的线程调用一次。用户可以使用 DisableThreadLibraryCalls()函数关闭线程终止时的通知。

不管是哪种情况下调用 DLL 入口，同一时间只能有一个线程调用入口函数。系统在调用函数的进程或线程上下文中调用入口函数。允许 DLL 使用自己的入口函数在调用进程的虚拟地址空间中分配内存，或者是打开可访问进程的句柄。入口函数也可以私自为使用线程本地存储（TLS）的新线程分配内存。

DLL 入口函数必须使用标准调用规范声明。在入口函数体中，读者可以处理 DLL 入口函数被调用情况的任何组合。一般情况下，入口函数应该只完成简单的初始化任务，如建立线程本地存储（TLS），创建同步对象和打开文件。在入口函数中不能调用 LoadLibrary()函数，因为可能在 DLL 装载顺序中产生依赖循环，导致 DLL 在系统执行初始化代码前使用。同样，在入口函数中也不能调用 FreeLibrary()函数，因为这可能导致 DLL 在系统已经执行终止代码后被使用。

调用除了 TLS 的 Win32 函数，同步和文件函数也可能导致难诊断的问题。如调用 User、Shell 和 COM 函数会导致访问违反错误，因为在这些 DLL 中，一些函数调用 LoadLibrary()函数装载其他系统组件。下面代码列出了 DLL 入口函数的结构。

```
01  BOOL APIENTRY DllMain( HANDLE hModule, DWORD ul_reason_for_call,
02                         LPVOID lpReserved )
03  {
04      switch (ul_reason_for_call)
05      {   //根据调用 DLL 的来源完成相应的工作
06      case DLL_PROCESS_ATTACH:
07          //对于每个新进程，初始化一次。如果 DLL 装载失败，则返回 false
08      case DLL_THREAD_ATTACH:
09          //执行线程指定初始化
10      case DLL_THREAD_DETACH:
11          //执行线程指定的清除工作
12      case DLL_PROCESS_DETACH:
13          //执行任何需要的清除工作
14          break;
15      }
16      return true;        //进程装载入口函数成功完成
17  }
```

当 DLL 入口函数在进程装载时调用，函数返回 true 表示成功。对于进程使用装载时链接，返回值 false 引起进程初始化失败和进程终止。对于进程使用运行时链接，返回值 false 表示 LoadLibrary()或 LoadLibraryEx()函数返回 NULL，表示失败。其他调用入口函数的情况，返回值可以忽略。

22.2.3　应用程序链接 DLL

使用装载动态链接的进程，当进程启动时，如果需要的 DLL 没有找到，系统会终止进

程，并提供给用户一个错误信息。此种情况下，系统不会终止使用运行时动态链接的进程，但是 DLL 导出的函数对应用程序来说是不可用的。有两种方法可以调用 DLL 中的函数。

- 装载时动态链接，模块清楚地调用导出 DLL 函数。此时，需要导入 DLL 库链接到模块。应用程序装载后，导入库提供装载 DLL 和定位导出 DLL 函数需要的信息的系统。
- 运行时动态链接。在模块运行时，使用 LoadLibrary()或 LoadLibraryEx()函数装载 DLL。DLL 装载后，模块调用 GetProcAddress()函数获取导出 DLL 函数的地址。模块使用 GetProcAddress()函数返回的函数指针调用导出 DLL 函数，去掉了导入库的工作。

当系统启动使用装载时动态链接的应用程序时，使用文件中的信息定位需要的 DLL 的名称。系统查找 DLL 时按照下面的顺序查找。

（1）应用程序装载的目录。

（2）当前目录。

（3）Windows 系统目录，使用 GetSystemDirectory()函数获取的目录路径。

（4）Windows 目录，使用 GetWindowsDirectory()函数获取的目录路径。

（5）在 PATH 环境变量中列出的目录。

如果系统不能查找到指定的 DLL，则终止进程，并显示一个报告错误对话框。否则，系统会映射 DLL 模块到进程的虚拟地址空间中，并增加 DLL 引用数量。接着系统调用入口函数。函数接收指示进程正在装载 DLL 的代码。如果入口函数没有返回 true，系统会终止进程并报告错误。最后，系统会修改进程代码，为引用的 DLL 函数提供开始地址。

在初始化时，DLL 被映射到进程的虚拟地址空间中，并且只有当需要时才会将其装载进物理内存中。单个应用程序调用 LoadLibrary()或 LoadLibraryEx()函数时，系统会使用与装载时动态链接使用的查找顺序相同的顺序试图查找 DLL。如果查找到了，系统映射 DLL 模块到进程的虚拟地址空间，并增加引用数量。如果使用 LoadLibrary()或 LoadLibraryEx()函数调用的 DLL，代码已经映射到调用进程的虚拟地址空间中，函数返回 DLL 句柄，并增加 DLL 的引用数量。注意两个具有相同基本文件名和扩展名但是不在同一个目录中的 DLL，作为两个不同的 DLL 对待。

系统在调用 LoadLibrary()或 LoadLibraryEx()的线程的上下文中调用入口函数。如果 DLL 已经被进程通过 LoadLibrary()或 LoadLibraryEx()装载，并且没有调用相应的 FreeLibrary()函数时，则系统不会调用入口函数。如果系统没有找到 DLL 或者入口函数返回 false，则 LoadLibrary()或 LoadLibraryEx()返回 NULL。如果 LoadLibrary()或 LoadLibraryEx()调用成功，则返回 DLL 模块的句柄。进程可以在 GetProcAddress()、FreeLibrary()或者 FreeLibraryAndExitThread()函数中使用句柄标识 DLL。

22.2.4　动态链接库函数

Win32 中提供了有关动态链接库的函数，下面做个简要介绍。DisableThreadLibraryCalls()函数关闭 hLibModule 句柄指定的动态链接库的 dll_thread_attach 和 dll_thread_detach 的通知消息。可以减少一些应用程序的工作代码区的大小。函数原型为：

```
BOOL DisableThreadLibraryCalls(
  HMODULE hLibModule);      //hLibModule 参数指定了要关闭消息的模块
```

如果 hLibModule 指定的 DLL 有活动的静态线程局部存储区，或者 hLibModule 是一个无效的模块句柄，则函数会失败。要获取具体的错误原因，可以通过调用 GetLastError()函数获得。

DllMain 是库定义函数名称的占位符，是 DLL 可选的入口方法。早期的 SDK 使用 DllEntryPoint 作为入口函数名。当构建 DLL 时，必须指定使用的实际名称。当进程和线程初始化或终止，或调用 LoadLibrary()和 FreeLibrary()函数时，系统会调用此函数。函数原型是：

```
BOOL WINAPI DllMain(
  HINSTANCE hinstDLL,    //DLL 模块的句柄
  DWORD fdwReason,       //调用函数的来源
  LPVOID lpvReserved     //预留
);
```

其中，fdwReason 参数指定 DLL 入口函数被调用的来源，取值如表 22-1 所示。

表 22-1　入口函数来源

值	含　义
Dll_process_attach	表示启动进程或调用 LoadLibrary()函数后，DLL 已经装载进当前进程的虚拟地址空间。DLL 可以在此初始化任何实例数据，或者使用 TlsAlloc()函数分配线程局部存储索引
Dll_thread_attach	表示当前进程创建新线程。当此时发生时，系统调用当前进程中的所有的 DLL 的入口函数。此调用在新线程的上下文中。DLL 可以在此初始化线程的 TLS 位置。使用 dll_process_attach 调用 DLL 入口函数的线程不会使用 dll_thread_attach 调用 DLL 入口函数。注意，使用此值调用 DLL 的入口函数仅仅会被进程装载 DLL 后创建的线程。当 DLL 使用 LoadLibrary 装载后，已经存在的线程不会调用新装载的 DLL 的入口函数
Dll_thread_detach	表示线程退出。如果 DLL 在 TLS 位置分配内存指针，在此释放内存。系统使用此值调用所有当前装载的 DLL 的入口函数。此调用是在退出线程的上下文中调用
Dll_process_detach	表示进程退出或调用 FreeLibrary()函数时，DLL 从调用进程的虚拟地址空间中卸载。DLL 可以在此调用 TlsFree()函数释放任何使用 TlsAlloc 分配的 TLS 索引，并释放任何线程局部数据

当系统使用 dll_process_attach 值调用 DllMain()函数时，如果成功返回 true，否则返回 false。当进程使用 LoadLibrary()函数调用 DllMain()函数时，如果返回值为 false，则 LoadLibrary()函数返回 NULL。当在进程初始化时调用 DllMain()函数时，如果返回值为 false，则进程会报告错误并终止进程。要获取更多的错误信息，可以使用 GetLastError()函数获取。

FreeLibrary()函数减少装载的 DLL 模块的引用数目。当引用数目为 0 时，模块将从调用进程的地址空间中卸载，并且句柄不再有效。

```
BOOL FreeLibrary( HMODULE hLibModule)   //装载的库模块的句柄
```

每个进程为每个装载的库模块维护一个引用数目。每调用一次 LoadLibrary()函数，引用数目增加 1，每调用一次 FreeLibrary()函数引用数目减 1。因为装载时动态链接有一个引

用数目，DLL 模块在进程初始化装载。如果调用 LoadLibrary()函数装载相同的模块，则数目增加。

在卸载库模块以前，如果有入口函数，则系统使用 dll_process_detach 值调用 DLL 的 DllMain()函数使得 DLL 从进程中分离。这样使得 DLL 可以有机会清除当前进程分配的资源。在入口函数返回后，库模块从当前进程的地址空间中移除。

从 DllMain()函数中调用 FreeLibrary 是不安全的。调用 FreeLibrary()函数不会影响其他使用相同库模块的进程。FreeLibraryAndExitThread()函数会减少装载的 DLL 的引用数目一次，然后调用 ExitThread()函数终止调用线程。此函数没有返回值。此函数给使用动态链接库创建和执行的线程一个安全的卸载 DLL 并终止它们的时机。函数原型为：

```
VOID FreeLibraryAndExitThread(
  HMODULE hLibModule,          //要减少引用次数的动态链接库
  DWORD dwExitCode);           //线程退出代码
```

GetModuleFileName()函数返回执行文件包含的指定模块的完整路径和文件名。函数原型为：

```
DWORD GetModuleFileName(
  HMODULE hModule,             //要获取文件名的模块句柄
  LPTSTR lpFilename,           //接收模块路径的缓冲区的指针
  DWORD nSize );               //缓冲区的大小
```

如果模块装载到两个进程中，在一个进程中的模块名称可能会与在另一个进程中的模块名称不同。如果文件已经映射到调用进程的地址空间中，则 GetModuleHandle()函数返回指定模块的模块句柄。函数原型为：

```
HMODULE GetModuleHandle(
  LPCTSTR lpModuleName)                //要获取句柄的模块的名称地址
```

如果函数成功，则返回值为指定模块的句柄；如果函数失败，返回值为 NULL。要获取更多的错误信息，可以调用 GetLastError()函数获取。

GetProcAddress()函数返回导出的 DLL 函数的地址。函数原型为：

```
FARPROC GetProcAddress(
  HMODULE hModule,                     //DLL 模块的句柄
  LPCSTR lpProcName);                  //函数名
```

LoadLibrary()函数映射指定的可执行模块到调用进程的地址空间中。其函数原型为：

```
HINSTANCE LoadLibrary(
 LPCTSTR lpLibFileName);               //可执行模块的文件名的地址
```

如果函数成功，则返回模块的句柄；如果函数失败，则返回值为 NULL。要获取更多的错误信息，可以通过 GetLastError()函数获取。

22.2.5　从动态库中获取位图资源

从动态库中获取位图资源的第一步需要使用 LoadLibraryEx()函数加载动态库，通过此函数返回的句柄对动态库中的资源进行处理。第二步需要调用 EnumResourceNames()函数

枚举指定类型的资源。此函数会查找模块中的指定类型的每个资源，并将当前枚举出的资源名称传递给应用程序定义的回调函数。其原型为：

```
BOOL EnumResourceNames(
  HINSTANCE hModule,              //指定要枚举资源的可执行文件的模块句柄
  LPCTSTR lpszType,               //指定要枚举的资源类型
  ENUMRESNAMEPROC lpEnumFunc,     //指定查找到每个资源后都要执行的回调函数
  LONG lParam);                   //传递给回调函数的用户自定义参数值
```

其中，lpszType 参数指定要枚举的资源类型，有效取值如表 22-2 所示。

表 22-2　可以枚举的资源的类型

值	含　义	值	含　义
RT_ACCELERATOR	加速键表资源	RT_GROUP_ICON	硬件支持的位图资源
RT_ANICURSOR	光标资源	RT_HTML	HTML 文档
RT_ANIICON	图标资源	RT_ICON	独立于硬件的位图资源
RT_BITMAP	位图资源	RT_MENU	菜单资源
RT_CURSOR	独立于硬件的光标资源	RT_MESSAGETABLE	消息表资源
RT_DIALOG	对话框资源	RT_PLUGPLAY	即插即用资源
RT_FONT	字体资源	RT_RCDATA	应用程序定义资源
RT_FONTDIR	字体目录资源	RT_STRING	字符表资源
RT_GROUP_CURSOR	硬件支持的光标资源	RT_VERSION	版本资源

其中回调函数的格式如下：

```
BOOL CALLBACK EnumResNameProc(
  HANDLE hModule,         //枚举函数正在枚举的资源所在的可执行文件的句柄
  LPCTSTR lpszType,       //正在枚举创建当前进程的模块的资源
  LPTSTR lpszName,        //当前枚举项的资源名称
  LONG lParam); //EnumResourceNames()函数的 lParam 参数传进来的用户自定义参数值
```

如果需要继续枚举，则函数应该返回 true；如果要停止枚举，则应该返回 false。下面是使用这些函数枚举位图资源的实例。

```
01  void CDLLAppSampleDlg::OnButtonGetbitmap()      //枚举位图资源的处理函数
02  {
03      m_iconList.ShowWindow(SW_HIDE);             //显示窗口
04      if( (hLibrary = LoadLibraryEx( "MORICONS.DLL", NULL,
05          LOAD_LIBRARY_AS_DATAFILE )) == NULL )   //装载动态库
06      {
07          WriteLog("文件载入错误!");
08          return;
09      }
10      //枚举资源名称
11      if(!EnumResourceNames(hLibrary,RT_BITMAP,
12          (ENUMRESNAMEPROC)EnumBitmapProcedure,(LPARAM)GetSafeHwnd()))
13          WriteLog("列举位图资源停止!");             //输出提示信息
14      FreeLibrary(hLibrary);                      //释放动态链接库
15  }
16  BOOL CALLBACK EnumBitmapProcedure(HANDLE hModule, LPCTSTR lpszType,
17      LPTSTR lpszName, LONG lParam)               //枚举回调函数
18  {
19      HBITMAP bitmap = LoadBitmap((HINSTANCE)hModule,lpszName);
```

```
20      SendMessage((HWND)lParam, WM BITMAP MESSAGE,
21          (LPARAM)bitmap, (WPARAM)lpszName);      //发送消息
22      return false;                               //返回 false
23  }
24  void CDLLAppSampleDlg::OnBitmapMessage(WPARAM wParam,LPARAM lParam)
25  {
26      Cstatic* m_Bitmap = (Cstatic*)GetDlgItem(IDC_STATIC_BITMAP);
27      //定义静态控件变量
28      m_Bitmap->SetBitmap((HBITMAP)wParam);       //装载位图
29      WriteLog((const char*)&lParam);             //输出提示信息
30  }
```

上面代码首先使用 LoadLibraryEx()函数装载 MORICONS.DLL，返回 DLL 的句柄，然后调用 EnumResourceNames()函数枚举位图资源。此函数中的第三个参数指定了回调函数，即每次查找到一个位图资源时要执行的函数，即 EnumBitmapProcedure()函数。在回调函数中，通过传入的实例句柄和资源名称，使用 LoadBitmap()函数装载位图资源，并将位图资源通过消息发送给主对话框。这里可以看到，返回值为 false，表示不再继续枚举。主对话框接收消息，并进入 OnBitmapMessage()函数，将 wParam 参数传入的位图资源句柄，显示在静态文本框中。程序运行效果如图 22-4 所示，显然此动态库中没有位图资源。

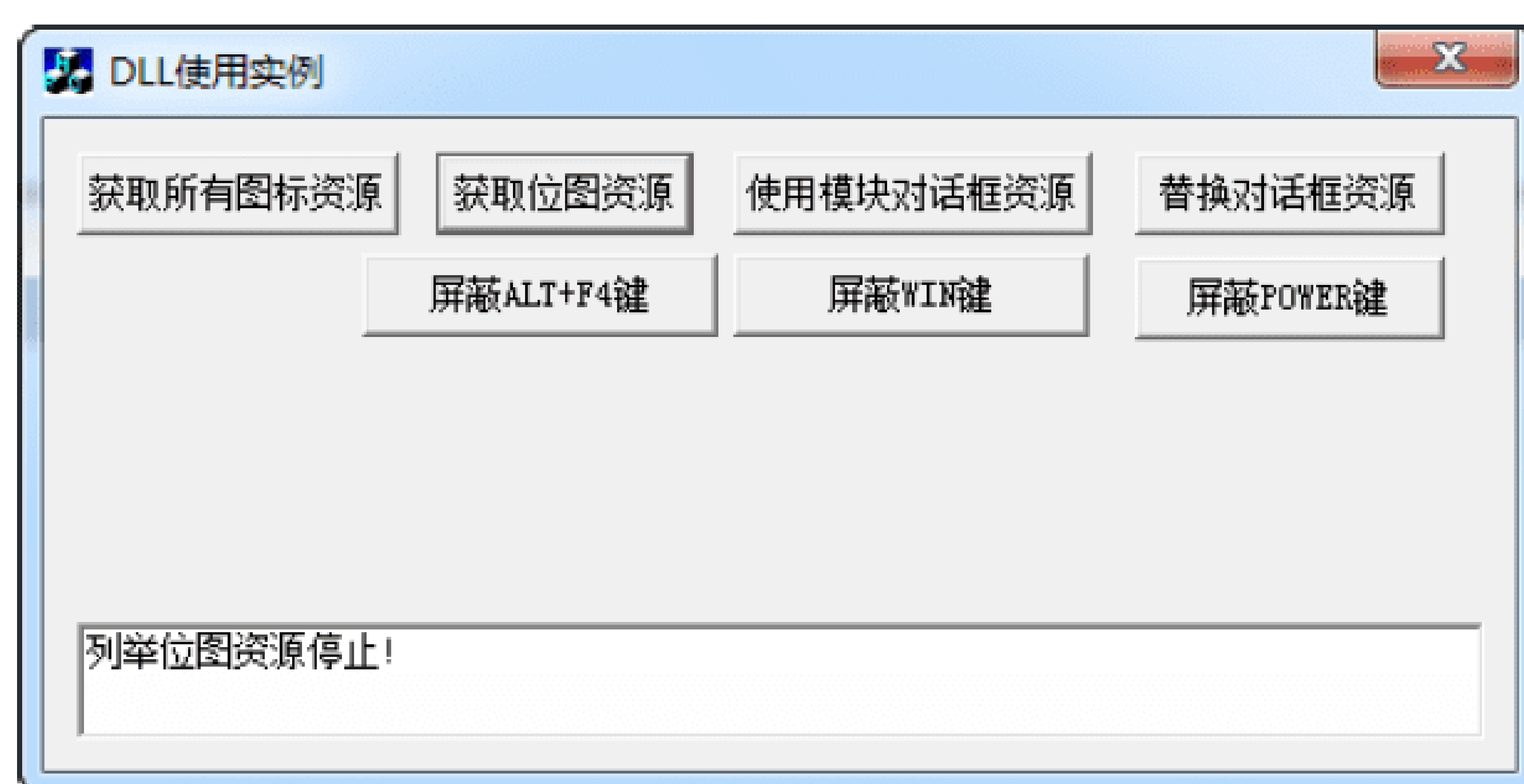


图 22-4　从动态库中获取位图资源

22.2.6　枚举模块中的所有图标

与从动态库中获取位图资源的过程一样，可以枚举模块中的所有图标。在本小节中会枚举所有的图标。只需将枚举函数的回调函数中的返回值变为 true 即可。代码如下。

```
01  void CDLLAppSampleDlg::OnButtonGetallicon()     //枚举图标
02  {
03      ResetContent();                             //清空显示控件
04      HINSTANCE  hLibrary;                        //静态库实例
05      if( (hLibrary = LoadLibraryEx( "MORICONS.DLL", NULL,
06          LOAD_LIBRARY_AS_DATAFILE )) == NULL )   //装载动态库
07      {
08          WriteLog("文件载入错误!");
09          return;
10      }
11      if(!EnumResourceNames(hLibrary,RT GROUP ICON,
12          (ENUMRESNAMEPROC)EnumIconProcedure,(LPARAM)GetSafeHwnd()))
13          //枚举资源
```

```
14          WriteLog("列举图标资源停止!");
15      FreeLibrary(hLibrary);                          //释放静态库
16      Cstring log;
17      log.Format("DLL 中共包含%d 个图标资源! ", m iconList.GetItemCount());
18      //格式化信息
19      WriteLog(log);                                  //输出信息
20  }
21  BOOL CALLBACK EnumIconProcedure(HANDLE hModule, LPCTSTR lpszType,
22      LPTSTR lpszName, LONG lParam)                   //枚举回调函数
23  {                                                   //装载图标
24      HICON icon = LoadIcon((HINSTANCE)hModule,lpszName);
25      SendMessage((HWND)lParam, WM_ICON_MESSAGE,(LPARAM)icon,(WPARAM)
26          lpszName);
27      return true;                                    //返回，继续枚举
28  }
29  void CDLLAppSampleDlg::OnIconMessage(WPARAM wParam,LPARAM lParam)
30  {
31      //显示图标
32      int iIconRet = imagelist.Add((HICON)wParam);//将位图添加到列表中
33      if (iIconRet!=-1)
34      {
35          //设置位图列表对象
36          m_iconList.SetImageList (&imagelist,LVSIL_SMALL);
37          int iIndex = m_iconList.GetItemCount();
38          //插入新项
39          m_iconList.InsertItem(iIndex, (const char*)&lParam, iIndex);
40      }
41  }
```

在上面代码中，OnButtonGetallicon()函数首先调用 LoadLibraryEx()函数装载 MORICONS.DLL，返回 DLL 的句柄，然后调用 EnumResourceNames()函数枚举图标资源。此函数中的第三个参数指定了回调函数，即每次查找到一个图标资源时，要执行的函数，即 EnumIconProcedure()函数。在回调函数中，通过传入的实例句柄和资源名称，使用 LoadIcon()函数装载图标资源，并将图标资源句柄通过消息发送给主对话框。主对话框接收消息，并进入 OnIconMessage()函数，将 wParam 参数传入的图标资源句柄，增加到 CimageList 变量中，并将其与列表控件关联起来，然后将图标资源的名称和图标显示出来。程序运行效果如图 22-5 所示。



图 22-5　枚举模块中的所有图标

22.2.7　使用模块对话框资源

通过 AfxSetResourceHandle()函数可以切换要使用的资源所在的实例，从而实现调用其他模块中的对话框资源的功能。其函数原型为：

```
void AfxSetResourceHandle(
  HINSTANCE hInstResource );          //指定应用程序要载入的资源所在的 EXE 或
                                      //DLL 文件的模块句柄或实例
```

在使用此函数之前，需要调用 AfxGetInstanceHandle()函数保存原来的实例句柄。在使用完指定文件中的资源后，重新调用 AfxSetResourceHandle()函数恢复原来的实例句柄。如下代码演示了如何调用 RedDLL.exe 文件中的“关于”对话框。

```
01  void CDLLAppSampleDlg::OnButtonGetdialog() //调用其他文件中的“关于”对话框
02  {
03      HINSTANCE m_hInstOld=AfxGetInstanceHandle();    //获取实例句柄
04      HINSTANCE m_hInstNew = LoadLibrary("RedDLL.exe");//装载 exe 文件
05      if (m_hInstNew == NULL)
06          //如果失败，则返回
07          WriteLog("装载可执行文件失败！");
08      AfxSetResourceHandle(m_hInstNew);               //设置资源句柄
09      Cdialog* dlg = new Cdialog();                    //创建对话框
10      if (dlg->Create(IDD_ABOUTBOX))
11          //显示对话框
12          dlg->ShowWindow(SW_SHOW);
13      AfxSetResourceHandle(m_hInstOld);               //设置资源句柄
14  }
```

上面代码首先调用 AfxGetInstanceHandle()函数保存当前的资源实例句柄，然后调用 LoadLibrary()函数装载 RedDLL.exe 程序，并通过 AfxSetResourceHandle()函数设置当前的资源实例句柄为载入的 RedDLL.exe 程序的实例句柄，表示现在开始使用 RedDLL.exe 中的资源。从中创建关于对话框，并显示。退出对话框后，重新调用 AfxSetResourceHandle()函数恢复应用程序原来的资源实例句柄。程序运行效果如图 22-6 所示。

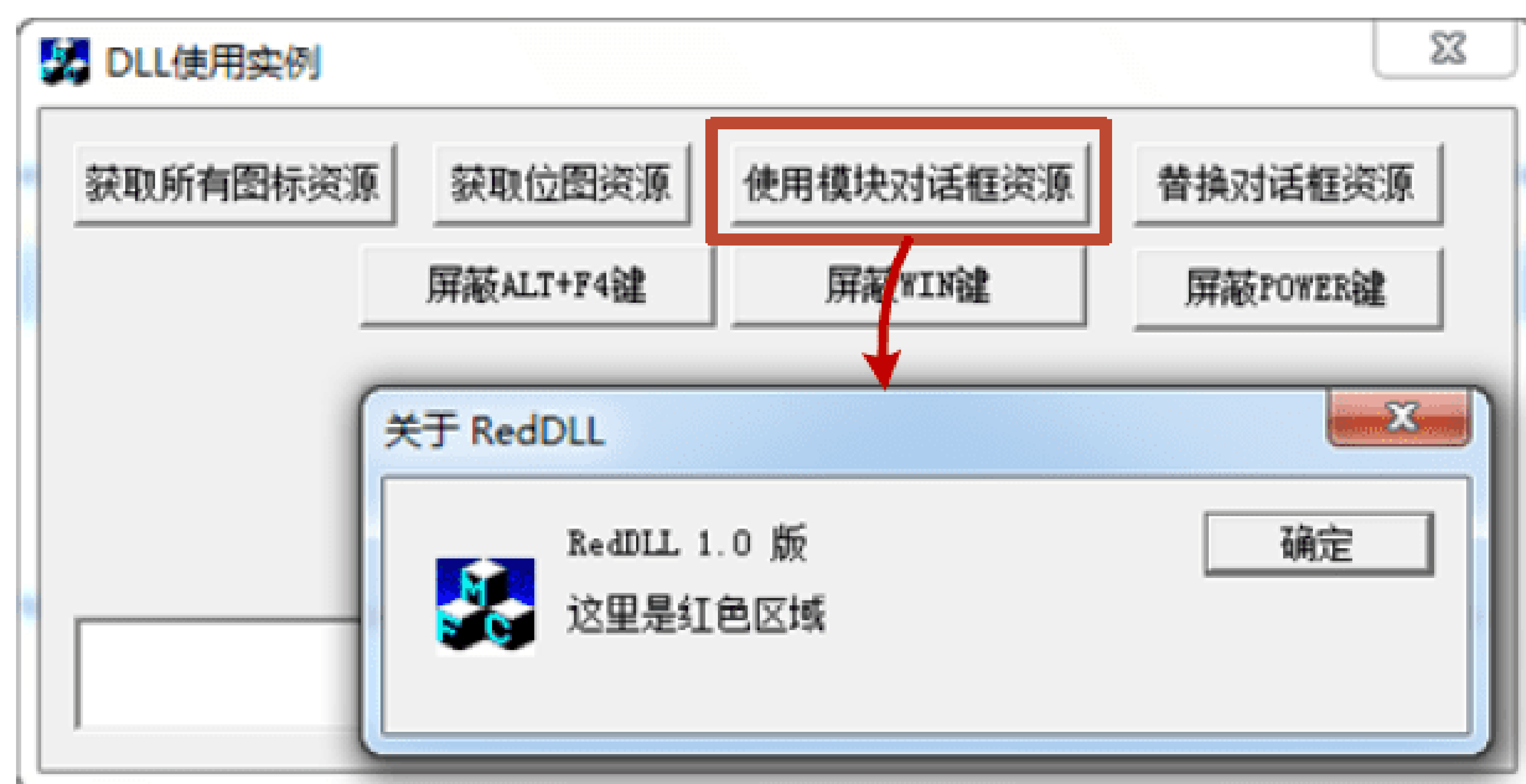


图 22-6　使用模块对话框资源运行效果

22.2.8　替换应用程序的对话框资源

Windows API 中提供了一组有关资源操作的函数，使用这组函数可以替换应用程序中

的对话框资源。过程分为以下几个步骤。

（1）使用 LoadLibrary()函数装载替换内容的可执行文件。

（2）使用 FindResource()函数和 LoadResource()函数查找定位并装载用于替换对话框资源。

（3）调用 LockResource()函数获取对话框资源的数据指针。

（4）使用 BeginUpdateResource()函数打开要更新的资源。

（5）使用 UpdateResource()函数将用于替换的对话框资源复制到要替换的对话框资源。

（6）使用 EndUpdateResource()函数完成替换。

下面以一个实例说明如何实现应用程序对话框的替换功能。在本例中，有两个应用程序，即 RedDLL.exe 和 GreenDLL.exe，RedDLL.exe 的“关于”对话框显示“这里是红色区域”，而 GreenDLL.exe 的“关于”对话框显示“这里是绿色区域”。程序要做的工作就是将 GreenDLL.exe 中的“关于”对话框使用 RedDLL.exe 中的“关于”对话框替换。代码如下：

```
01  void CDLLAppSampleDlg::OnButtonReplacedialog() //替换对话框资源
02  {
03      HRSRC hRes, hResLoad;                       //资源句柄
04      HANDLE hExe, hUpdateExe;                    //句柄
05      char *lpResLock;                            //资源锁
06      BOOL bResult;                               //定义返回值
07      hExe = LoadLibrary("RedDLL.exe");           //装载 RedDLL
08      if (hExe == NULL)
09          WriteLog("装载可执行文件失败！");
10      hRes = FindResource((HINSTANCE)hExe,
11          MAKEINTRESOURCE(IDD_ABOUTBOX), RT_DIALOG);//查找资源
12      if (hRes == NULL)
13          WriteLog("无法查找要替换的资源！");
14      hResLoad = (HRSRC)LoadResource((HINSTANCE)hExe, hRes);
15      if (hResLoad == NULL)
16          WriteLog("无法装载对话框！");
17      lpResLock = (char*)LockResource(hResLoad); //锁定资源
18      if (lpResLock == NULL)
19          WriteLog("无法锁定对话框！");
20      hUpdateExe = (HRSRC)BeginUpdateResource("GreenDLL.exe",false);
21      //装载 GreenDLL
22      if (hUpdateExe == NULL)
23          WriteLog("无法打开要写入资源的文件！");
24      bResult = UpdateResource((HINSTANCE)hUpdateExe, RT_DIALOG,
25          MAKEINTRESOURCE(IDD_ABOUTBOX),
26          MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL), lpResLock,
27          SizeofResource((HINSTANCE)hExe, hRes)); //替换资源
28      if (bResult == false)
29          WriteLog("替换资源失败！");
30      if (!EndUpdateResource(hUpdateExe, false))
31          WriteLog("不能写入对文件的修改！");
32      if (!FreeLibrary((HINSTANCE)hExe))
33          WriteLog("释放文件失败！");
34  }
```

除了前面提及的几个函数的使用，还要注意 MAKEINTRESOURCE 宏和 MAKELANGID 宏的使用，这些宏，实现从已知的宏转换成实际需要的值。程序运行的效果如图 22-7 所示。其中，第一幅图是替换对话框资源之前单击 GreenDLL.exe 程序的“关

于”菜单项时，显示的对话框，其中显示的是“这里是绿色区域”；而第二幅是替换对话框资源之后单击 GreenDLL.exe 程序的“关于”菜单项时，显示的对话框，其中显示的是“这里是红色区域”。

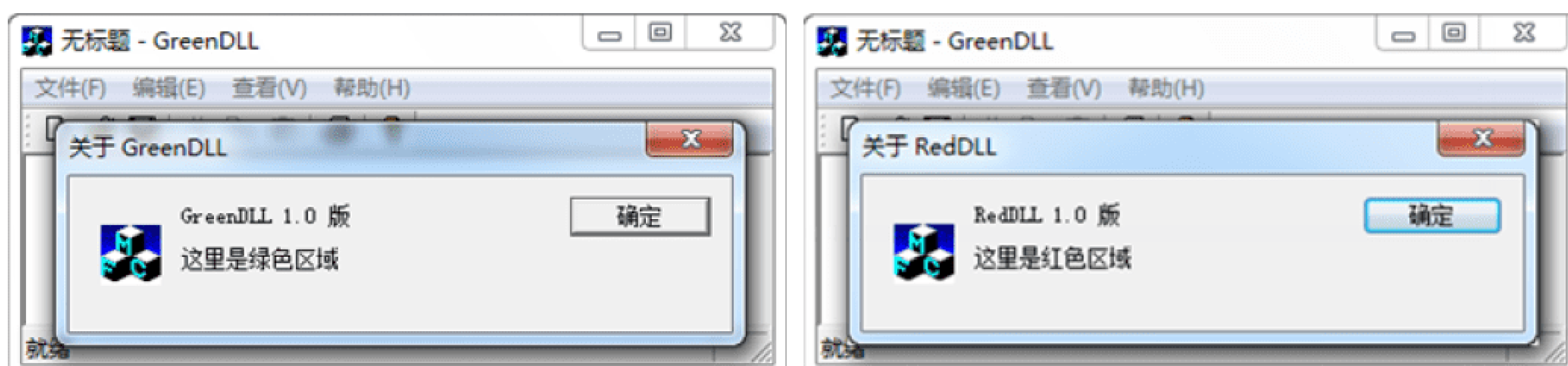


图 22-7　替换应用程序对话框运行效果

22.2.9　屏蔽键盘 Power 键

通过回调函数，可以屏蔽系统按键的处理。本小节介绍通过底层键盘钩子的回调函数来屏蔽 POWER 键的方法。键盘钩子是当有键盘事件发生时，系统会执行的操作，操作执行的内容可以定义在回调函数中。读者可以通过挂上相应的钩子来自定义操作。由于 POWER 键是底层键盘按键，因此需要使用底层键盘钩子回调函数。其原型为：

```
LRESULT CALLBACK LowLevelKeyboardProc(
  int nCode,          //指定钩子处理代码
  WPARAM wParam,      //指定键盘消息的标识符
  LPARAM lParam);     //指向 KBDLLHOOKSTRUCT 结构的指针，其中存储按下的按键的信息
```

如果 nCode 参数为 HC_ACTION，则在 wParam 和 lParam 参数中包含与键盘消息相关的信息。如果 nCode 小于 0，则钩子处理函数必须通过 CallNextHookEx()函数将消息传递给下一个钩子。wParam 参数用于指定键盘消息的标识符，取值为 WM_KEYDOWN、WM_KEYUP、WM_SYSKEYDOWN 或 WM_SYSKEYUP，分别表示按键按下、按键抬起、系统按键按下和系统按键抬起。如果按下的按键是要屏蔽的按键，则返回 true，否则，返回 CallNextHookEx()函数。代码如下：

```
01  //底层键盘钩子回调函数
02  LRESULT CALLBACK LowLevelKeyboardProc(int nCode,
03                                 WPARAM wParam, LPARAM lParam)
04  {
05      if (nCode == HC_ACTION)                    //禁用键盘的某个按键
06      {
07          KBDLLHOOKSTRUCT* pHookStruct = (KBDLLHOOKSTRUCT*)lParam;
08          //取出钩子结构
09          LPDWORD tmpVirKey = m_lpdwVirtualKey;//取出要屏蔽的键的列表指针
10          for (int i = 0; i < m_nLength; i++) //依次处理要屏蔽的按键
11          {
12              if (*m_lpdwContentKey++ == 1)
13              {
14                  DWORD dwAltKey = 32;
15                  if ((pHookStruct->vkCode == *tmpVirKey++)&&
16                      (pHookStruct->flags == dwAltKey))
```

```
17                  return true;
18              }
19              if (pHookStruct->vkCode == *tmpVirKey++) return true;
20          }
21      }
22      //传给系统中的下一个钩子
23      return CallNextHookEx(m_hHook, nCode, wParam, lParam);
24  }
```

上面的代码是底层键盘钩子的回调函数。判断按下的功能组合键和按键是否与禁用的按键相同，如果是要屏蔽的按键，则返回 true，终止向下一个钩子函数传递，从而屏蔽到此按键的事件；如果不是要屏蔽的按键，则返回 CallNextHookEx，将消息传递给下一个钩子。为了使得功能可以重复使用，将此功能封装在 DLL 中，而需要屏蔽的程序只需要调用 DLL 即可。以下代码是屏蔽按键的处理过程。

```
01  SHIELDKEYBORDSAMPLE_API bool StartShieldKey(LPDWORD lpdwVirtualKey,
02      LPDWORD lpdwContentKey, int nLength)    //屏蔽按键处理
03  {
04      if (m_hHook != NULL)
05          return StopShieldKey();             //执行停止按键处理函数
06      m_lpdwVirtualKey = (LPDWORD)malloc(sizeof(DWORD) * nLength);
07      m_lpdwContentKey = (LPDWORD)malloc(sizeof(DWORD) ** nLength);
08      LPDWORD tmpVirKey = m_lpdwVirtualKey;   //要屏蔽的底层键盘
09      LPDWORD tmpConKey = m_lpdwContentKey;   //要屏蔽的底层键盘组合键
10      for (int i = 0; i < nLength; i++)
11      {
12          *tmpVirKey++ = *lpdwVirtualKey++;
13          *tmpConKey++ = *lpdwContentKey++;
14      }
15      m_nLength = nLength;
16      m_hHook = SetWindowsHookEx(WH_KEYBOARD_LL, LowLevelKeyboardProc,
17          m_hInstance, NULL);                 //安装底层键盘钩子
18      if (m_hHook == NULL)
19          return false;
20      return true;
21  }
22  SHIELDKEYBORDSAMPLE_API bool StopShieldKey()//卸载按键钩子
23  {
24      if (UnhookWindowsHookEx(m_hHook) == 0)
25          return false;
26      m_hHook = NULL;
27      return true;
28  }
```

在上面代码中，StartShieldKey()函数用于安装钩子。首先将传入的要屏蔽的按键的数组传入 DLL 中，并保存为全局变量。然后调用 SetWindowsHookEx()函数安装钩子，并指定钩子的回调函数为 LowLevelKeyboardProc()。StopShieldKey()函数用于卸载钩子。调用 UnhookWindowsHookEx()函数卸载指定的钩子。下面的代码就是在调用的程序中，调用函数安装屏蔽 Power 键的钩子。

```
01  void CDLLAppSampleDlg::OnButtonDisablePower()       //屏蔽 Power 按键
02  {
03      DWORD dwVerKey[] = {0x00000001};                //定义按键集合
04      DWORD dwConKey[] = {0};
05      int nLength = sizeof(dwVerKey) / sizeof(DWORD);//计算长度
```

```
06      if (StartShieldKey(dwVerKey, dwConKey, nLength))
07          WriteLog("已经屏蔽 Power 键");
08      else
09          WriteLog("屏蔽 Power 键失败");
10  }
```

因为 Power 键的按键代码是 1，因此将其传入 DLL 中的 StartShieldKey()函数后可以屏蔽 Power 按键。

22.2.10　屏蔽键盘 Win 键

与屏蔽 Power 按键一样，要屏蔽 Win 键，只需要将 Win 按键的按键代码传入即可。下面的代码会屏蔽键盘左边的 Win 键和右边的 Win 键。

```
01  void CDLLAppSampleDlg::OnButtonDisableWin()         //屏蔽 Win 按键
02  {
03      DWORD dwVerKey[] = {VK_LWIN, VK_RWIN};          //定义 Win 按键数组
04      DWORD dwConKey[] = {0, 0};
05      int nLength = sizeof(dwVerKey) / sizeof(DWORD);//计算长度
06      if (StartShieldKey(dwVerKey, dwConKey, nLength))
07          WriteLog("已经屏蔽了 Win 按键");
08      else
09          WriteLog("屏蔽 Win 按键失败");
10  }
```

22.2.11　禁止使用<Alt+F4>组合键关闭窗体

与屏蔽 Power 按键一样，要禁止使用<Alt+F4>组合键，只需要将 Alt 按键和 F4 按键传入即可。此处，是将按键和对应的组合功能键通过两个数组传入。代码如下：

```
01  void CDLLAppSampleDlg::OnButtonDisableAltf4()       //屏蔽 Alt+F4 按键
02  {
03      DWORD dwVerKey[] = {VK_F4};                     //定义按键数组
04      DWORD dwConKey[] = {1};
05      int nLength = sizeof(dwVerKey) / sizeof(DWORD);//计算按键长度
06      if (StartShieldKey(dwVerKey, dwConKey, nLength))
07          WriteLog("已经屏蔽了 Alt+F4 按键");
08      else
09          WriteLog("屏蔽 Alt+F4 按键失败");
10  }
```

上面代码中将 F4 键和功能按键 Alt 两个值传入 StartShieldKey()函数。程序运行后，使用<Alt+F4>组合键窗体不会执行任何操作。

22.3　MFC 常规 DLL 的创建与使用实例

22.2 节中介绍了非 MFC DLL 的创建和使用实例，与之不同的是，本节介绍内部使用 MFC，但是提供的访问接口不支持 DLL 而是标准的 C 接口的常规 DLL。除了介绍基本概念和创建方法外，本节还将介绍 MFC 常规 DLL 的创建和使用方法。

22.3.1　基本概念

MFC 常规 DLL，从字面上理解有两点。一是 MFC 的，这是指 DLL 内部使用 MFC 进行编程。二是指其是常规的，这是指此种 DLL 提供的接口是常规的而不是 DLL 的。从这种类型的 DLL 中导出的函数可以被 MFC 也可以被非 MFC 应用程序调用，从其中导出的函数使用标准的 C 接口。

MFC 常规 DLL 具有一个对应的 CWinApp 对象，并且初始化和析构任务与 MFC 应用程序的处理位置是相同的，分别在 DLL 的 CWinApp 派生类的 InitInstance()成员函数和 ExitInstance()成员函数中处理。因为 MFC 提供了 DllMain()函数，因此，不需要手动编写此函数。DllMain()函数在 DLL 装载时调用 InitInstance()函数，在 DLL 卸载时调用 ExitInstance()函数。MFC 常规 DLL 分为以下两种，分类标准是链接 MFC DLL 的方式。

- ❑ 静态链接 MFC 的规则 DLL，在内部使用 MFC，使用 MFC 的静态链接库生成 DLL。
- ❑ 动态链接 MFC 的规则 DLL，在内部使用 MFC 并动态链接到 MFC。使用此方式的规则 DLL，则必须在 DLL 的所有导出函数的开头使用 AFX_MANAGE_STATE 宏，设置当前模块状态为 DLL 中的一个。

22.3.2　创建 MFC 常规 DLL

在 Visual Studio 2010 中创建 MFC 常规 DLL 的步骤如下。

（1）打开 VC，选择“文件”|“新建”|“项目”命令，弹出“新建项目”对话框，如图 22-8 所示。

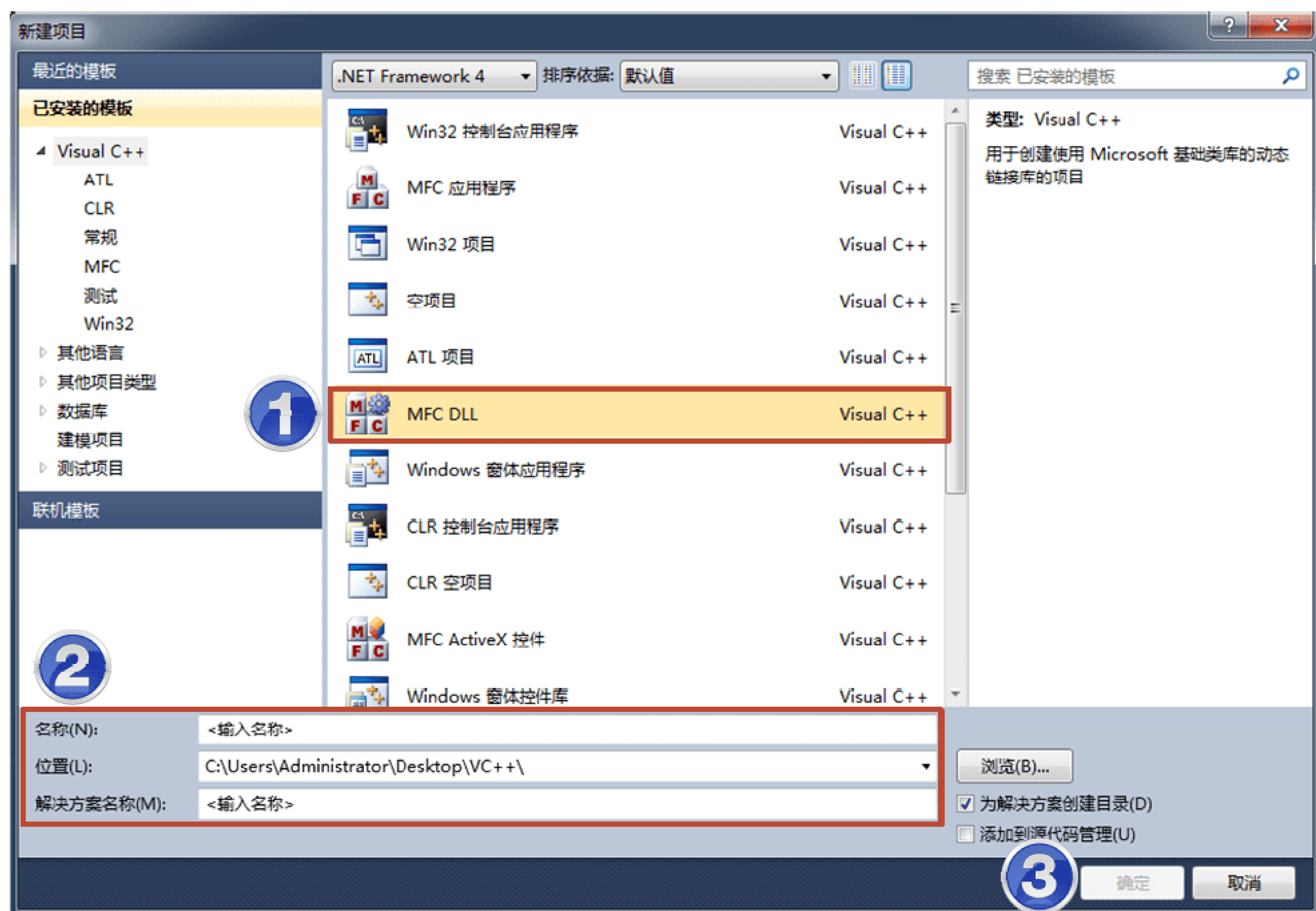


图 22-8　创建 MFC 常规 DLL 的第一步

（2）在“名称”文本框和“位置”文本框中填入相应的值，并选择 MFC DLL 图标。单击“确定”按钮，弹出 “MFC DLL 向导”对话框，如图 22-9 所示。

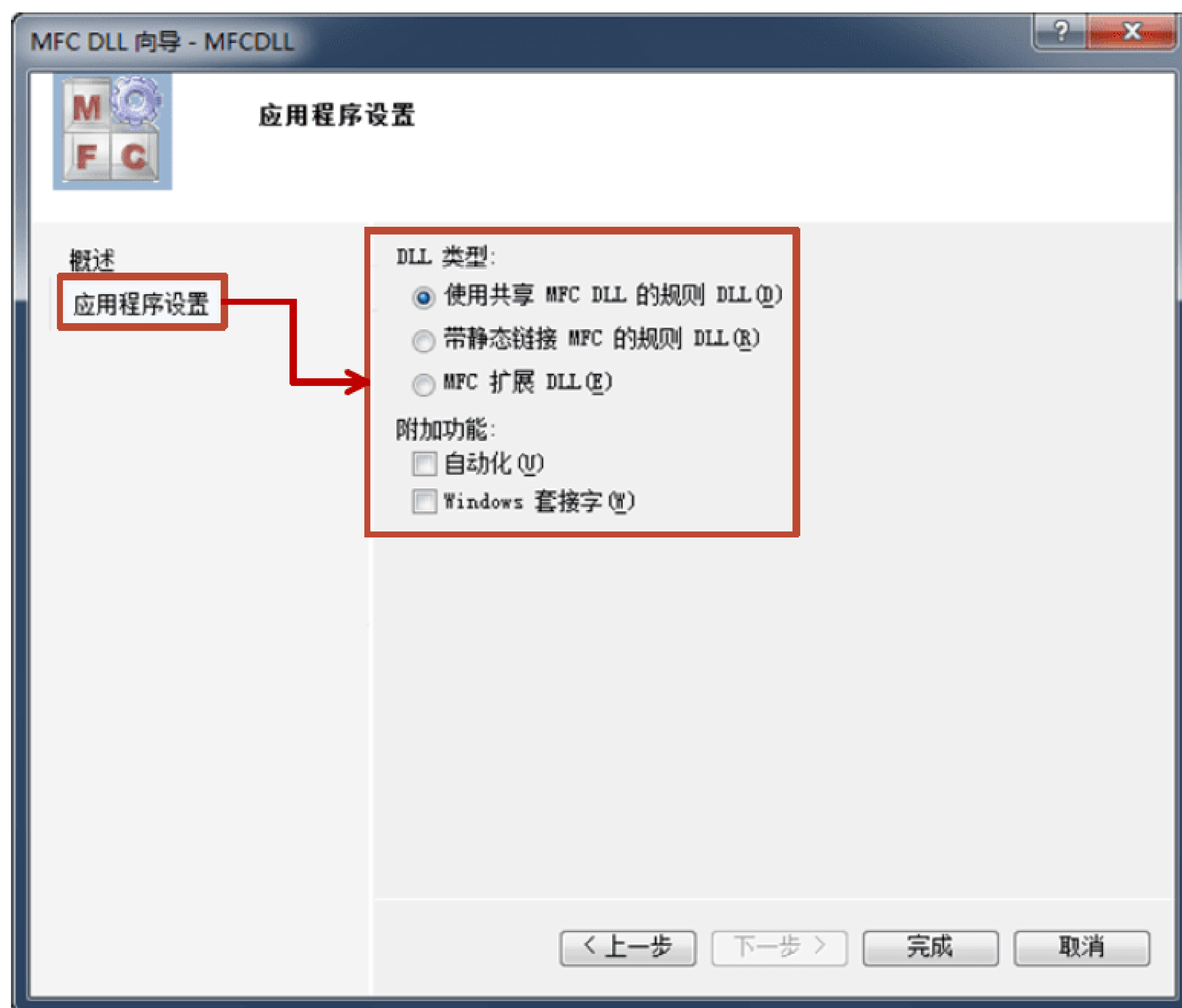


图 22-9　创建 MFC 常规 DLL 的第二步

（3）在“DLL 类型”选项组中选择创建的 DLL 种类。单击“完成”按钮。

（4）这样就成功地创建了一个 MFC 常规 DLL。

在创建了 DLL 后，就可以将程序功能添加到 DLL 中。

22.3.3　MFC 常规 DLL 的创建实例

22.3.2 小节介绍了创建 MFC 常规 DLL 的方法，本小节以一个实例讲解具体过程。本小节实例实现的功能是创建一个通过 MFC 实现的对话框类，并在导出的接口函数中调用此对话框。具体过程如下。

（1）按照 22.3.2 小节中介绍的方法，创建 MFC 常规 DLL。

（2）在 DLL 工程中按照前面讲过的方法，添加一个对话框资源，并为此对话框资源创建派生自 CDialog 类的对话框实例类，并在对话框内添加实现的功能。本实例中，实现单击对话框类中的按钮，则在静态框中显示欢迎词的功能。

（3）添加调用此对话框的接口函数。接口函数需要使用 extern"C"__declspec(dllexport)修饰符指定，使其作为导出接口函数。代码如下：

```
01  extern "C" __declspec(dllexport) void ShowDlg(void)//显示对话框
02  {
03      AFX_MANAGE_STATE(AfxGetStaticModuleState());
04      CdlgDllTest dlg;                                    //定义对话框变量
05      dlg.DoModal();                                      //显示对话框
06  }
```

需要注意的是，此处在调用 CdlgDllTest 对话框前，需要调用 AFX_MANAGE_STATE 宏，此宏的功能是进行模块状态的切换，而 AfxGetStaticModuleState()函数是在程序堆栈上创建一个 AFX_MODULE_STATE 类型的实例，以切换当前运行的模块状态。在动态链接 MFC 的常规 DLL 的每个接口函数中都需要调用此语句，或是在调用 DLL 的地方使用资源切换的方式（这种方式在第 22.2 节中介绍过）。不管哪种方式都需要进行运行程序状态切换，才可以完成对资源对话框的调用。

（4）添加完功能代码后，编译链接 DLL，生成 RegMFCDLLSample.dll 即可。

22.3.4　调用 MFC 常规 DLL

创建完 MFC 常规 DLL 后，就可以在应用程序中调用它了。MFC 常规 DLL 既可以被 MFC 应用程序调用，也可以被非 MFC 应用程序调用。调用 MFC 常规 DLL 的方式有两种，一种是静态引用，通过加载静态链接库的 lib 文件实现。一种是动态引用，动态加载 DLL 后，获取要调用的函数地址，然后执行相应的函数。本小节以动态加载的方式演示如何调用 MFC 常规 DLL。代码如下：

```
01  void CRegMFCDllTestDlg::OnButtonInvokedll()         //调用 DLL
02  {
03      typedef void (*pFunction)(void);                //定义函数变量
04      HINSTANCE hLibrary;                             //DLL 句柄
05      hLibrary = LoadLibrary("RegMFCDLLSample.dll"); //装载 DLL
06      if (hLibrary == NULL)
07          MessageBox("DLL 加载失败");    //提示错误信息
08      pFunction pShowDlg =
09           (pFunction)GetProcAddress(hLibrary,"ShowDlg");
10      //执行函数
11      if (NULL==pShowDlg)
12          MessageBox("DLL 中不存在指定的函数");            //输出提示
13      else
14          pShowDlg();
15  }
```

上面代码定义了函数指针变量 pFunction。调用 LoadLibrary()函数装载要执行的 DLL，此处是 RegMFCDLLSample.dll。如果加载成功，使用 GetProcAddress()函数获取要执行的接口函数的地址，如果查找到函数，则执行。程序运行的效果如图 22-10 所示。

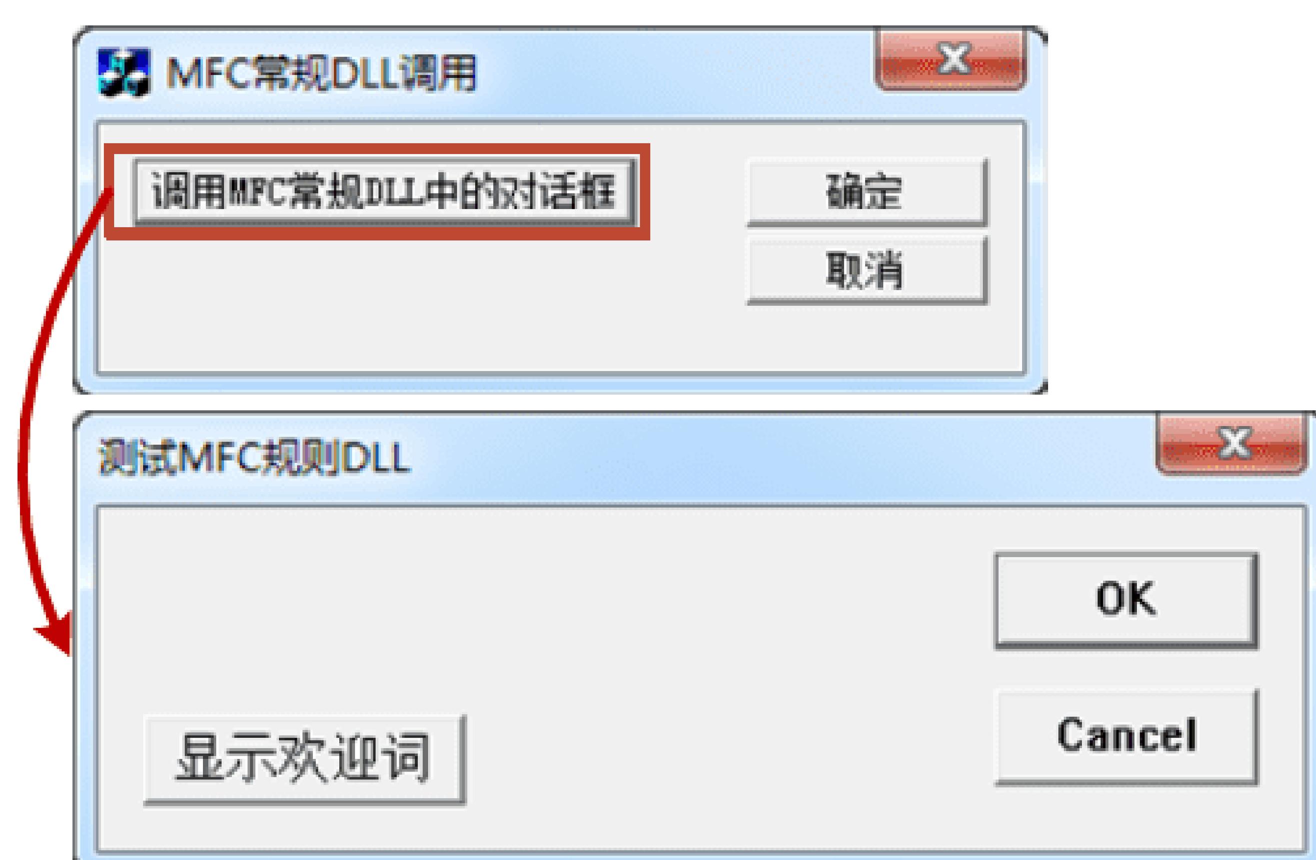


图 22-10　MFC 常规 DLL 调用实例运行效果

22.4　MFC 扩展 DLL 的创建与使用实例

MFC 常规 DLL 是使用 MFC 但是导出的接口不支持 MFC 的 DLL，而 MFC 扩展 DLL 则是内部既使用 MFC，导出的接口也支持 MFC 的 DLL，解决了要在 DLL 和 EXE 之间传递从 MFC 派生而来的类的问题。本节介绍 MFC 扩展 DLL 的创建和使用实例。

22.4.1　创建 MFC 扩展 DLL

MFC 扩展 DLL 实现继承子 MFC 类库中已经存在的类，完成可重复使用的类的 DLL。扩展 DLL 使用 MFC 的动态链接版本，也就是 MFC 共享版本。只有使用 MFC 共享版本生成的 MFC 可执行文件（应用程序或规则 DLL），才可以使用扩展 DLL。使用扩展 DLL 可以从 MFC 中继承新的自定义类，并为应用程序提供扩展的 MFC 版本。

- ❑ DLL 的客户端 EXE 必须是使用_AFXDLL 编译的 MFC 应用程序。
- ❑ 动态链接到 MFC 的规则 DLL 也可以使用扩展 DLL。
- ❑ 扩展 DLL 也可以使用_AFXEXT 定义进行编译，强制定义_AFXDLL，并保证正确的特性。使得当生成 DLL 时，AFX_EXT_CLASS 定义为__declspec(dllexport)，如果在扩展 DLL 中使用宏声明类则是必须的。
- ❑ 扩展 DLL 不会实例化从 CWinApp 继承的类，但是依赖于客户端应用程序或 DLL 提供对象。
- ❑ 扩展 DLL 也提供一个 DllMain()函数，并进行必要的初始化工作。

扩展 DLL 使用 MFC 的动态链接版本生成（也就是共享 MFC 版本）。只有使用共享版本 MFC 的 MFC 可执行程序（应用程序或规则 DLL），才可以使用扩展 DLL。无论是客户端应用程序还是扩展 DLL，必须使用相同的 MFC.DLL 版本。

扩展 DLL 可以在应用程序和 DLL 之间传递派生自 MFC 的对象。在对象创建的模块中与传入对象相关的成员函数也会传入。因为当使用共享 MFC 的 DLL 版本时，这些函数被正确地导出，可以在应用程序和导入的扩展 DLL 之间自由地传递 MFC 或派生的 MFC 对象指针。

MFC 扩展 DLL 使用共享版本的 MFC 与应用程序使用共享版本的 DLL 的方法是相同的，但是也有不同之处。

- ❑ 没有继承自 CWinApp 的派生类。必须与客户端应用程序的 CwinApp 派生对象一起工作。也就是说，客户端应用程序处理主消息队列、空闲队列等。
- ❑ 在 DllMain()函数中调用 AfxInitExtensionModule()函数，并检测此函数的返回值。如果此函数返回 0，则从 DllMain()函数中返回 0。
- ❑ 如果扩展 DLL 想导出应用程序的 CRuntimeClass 对象或资源，则会在初始化时，创建 CdynLinkLibrary 对象。

如果使用 DEF 文件导出，在头文件的开头和结尾处放置以下代码。

```
#undef AFX_DATA
#define AFX_DATA AFX_EXT_DATA
```

```
//头文件体
#undef AFX DATA
#define AFX_DATA
```

这 4 行代码可以保证扩展 DLL 中的代码正确编译。如果没有这 4 行，会导致 DLL 编译或链接不正确。创建 MFC 扩展 DLL 的方法与创建 MFC 常规 DLL 的步骤基本是相同的，只是在创建 MFC DLL 的第一步中，选择 DLL 的类型为 MFC Extension DLL（using）shared MFC DLL，则创建的 DLL 就是 MFC 扩展 DLL。

22.4.2　MFC 扩展 DLL 的创建实例

22.4.1 小节介绍了创建 MFC 扩展 DLL 的方法，本小节以一个实例讲解具体过程。本小节实例实现的功能是创建一个通过 MFC 实现的对话框类，并在一个导出类中提供调用此对话框的接口函数。具体过程如下。

（1）按照 22.4.1 小节中介绍的方法，创建 MFC 扩展 DLL。

（2）在 DLL 工程中按照前面讲过的方法，添加一个对话框资源，并为此对话框资源创建派生自 CDialog 类的对话框实例类。并在对话框内添加实现的功能。本实例中，实现单击对话框类中的按钮，在静态框中显示欢迎词。

（3）添加调用此对话框的接口类。接口类需要使用 AFX_EXT_CLASS 修饰符指定，使其作为导出类。代码如下：

```
class AFX EXT CLASS  CExtDLLClass : Cobject
{
    public:
        void ShowDlg();
        CExtDLLClass();
        virtual ~CExtDLLClass();
};
```

其中，在 ShowDlg()函数中会调用自定义的对话框 CDlgExtDLL。而自定义对话框类 CDlgExtDLL 可以按照普通的对话框程序一样设计使用。

（4）添加完功能代码后，编译链接 DLL，生成 ExtMFCDLLSample.dll 即可。

22.4.3　调用 MFC 扩展 DLL

创建完 MFC 扩展 DLL 后，就可以在应用程序中调用它了。MFC 扩展 DLL 既可以被 MFC 应用程序调用，也可以被非 MFC 应用程序调用。调用 MFC 扩展 DLL 的方法是通过静态引用，即通过加载静态链接库的 lib 文件实现。要完成对 MFC 扩展 DLL 的调用，需要 3 个资源。

- 包含要调用的类的头文件，在本例中是 ExtDLLClass.h 文件。
- 需要加载 MFC 扩展 DLL 对应的静态链接库 LIB 文件，在本例中是 ExtMFCDLLSample.lib 文件。
- MFC 扩展 DLL 的动态链接库，在本例中是 ExtMFCDLLSample.dll 文件。

下面代码表示调用 MFC 扩展 DLL 中的接口类提供的对话框功能。

```
01  void CExtMFCDLLTestDlg::OnButtonInvokedlg()     //调用 DLL 中的对话框
02  {
03      CExtDLLClass dlg;                           //定义对话框变量
04      dlg.ShowDlg();                              //显示对话框
05  }
```

从上面可以看出，在调用 MFC 扩展 DLL 的时候，调用方法与普通的 MFC 类调用的方式是相同的。在本例中，DLL 导出的类是继承自 MFC 的 Cobject 类，同样也可以导出派生自 MFC 的其他类。程序运行效果如图 22-11 所示。

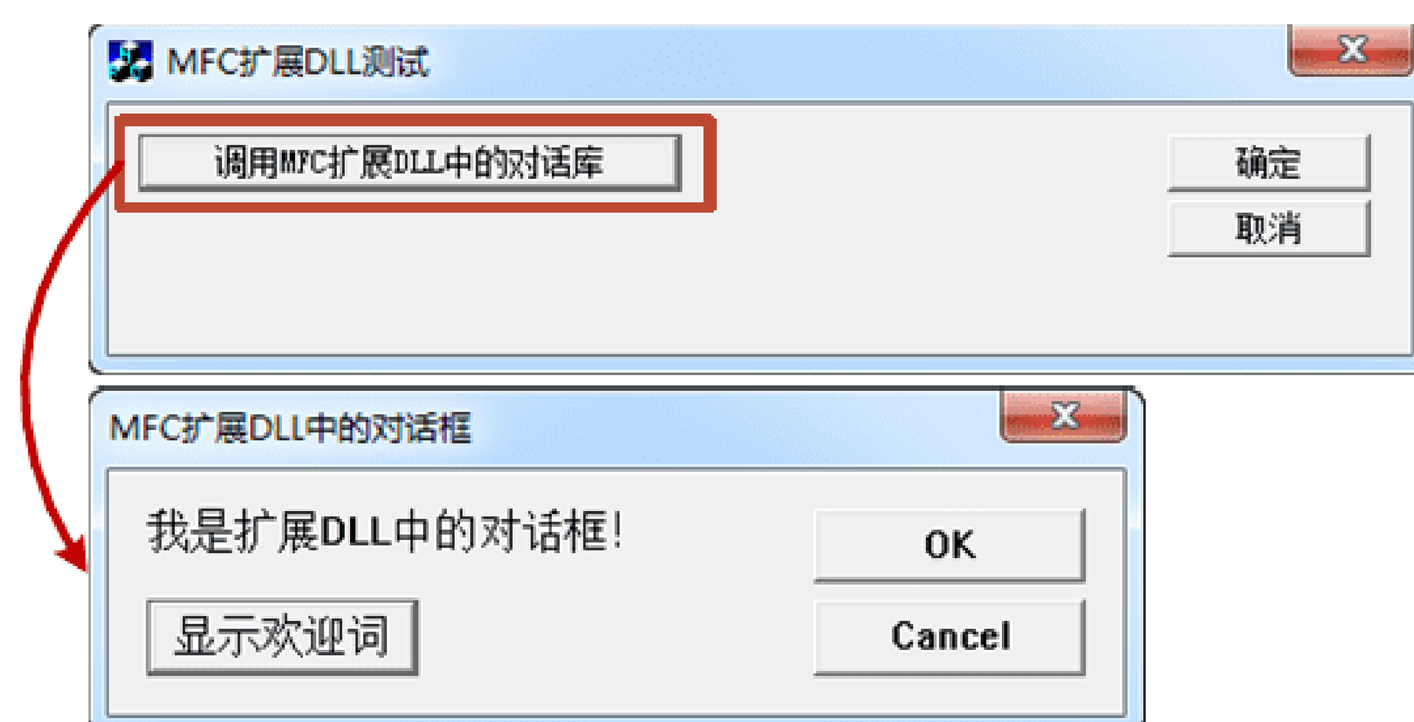


图 22-11　调用 MFC 扩展 DLL 的运行效果

22.5　DLL 的查看与调试

Windows 操作系统的核心功能是采用模块的方式实现的。它将各种相关功能放置在同一 DLL 模块中。因此，每个应用程序都会调用相关的系统的或用户自定义的 DLL。因此，在编写程序时，就必须掌握 DLL 的查看和调试的方法。

22.5.1　使用 Depends 工具查看 DLL 接口

Visual Studio 2010 不再提供查看 DLL 接口的工具——Depends。所以这里对 Visual C++ 6.0 提供的工具做简单介绍，调用方法是选择“开始”|“所有程序”|Microsoft Visual Studio 6.0|Microsoft Visual Studio 6.0 Tools|Depends 命令，即可打开 Depends 工具。要查看 DLL 接口，则选择 File|Open 命令，选择要查看的 DLL 文件，则界面中会显示调用的 DLL 及其提供的接口，如图 22-12 所示。

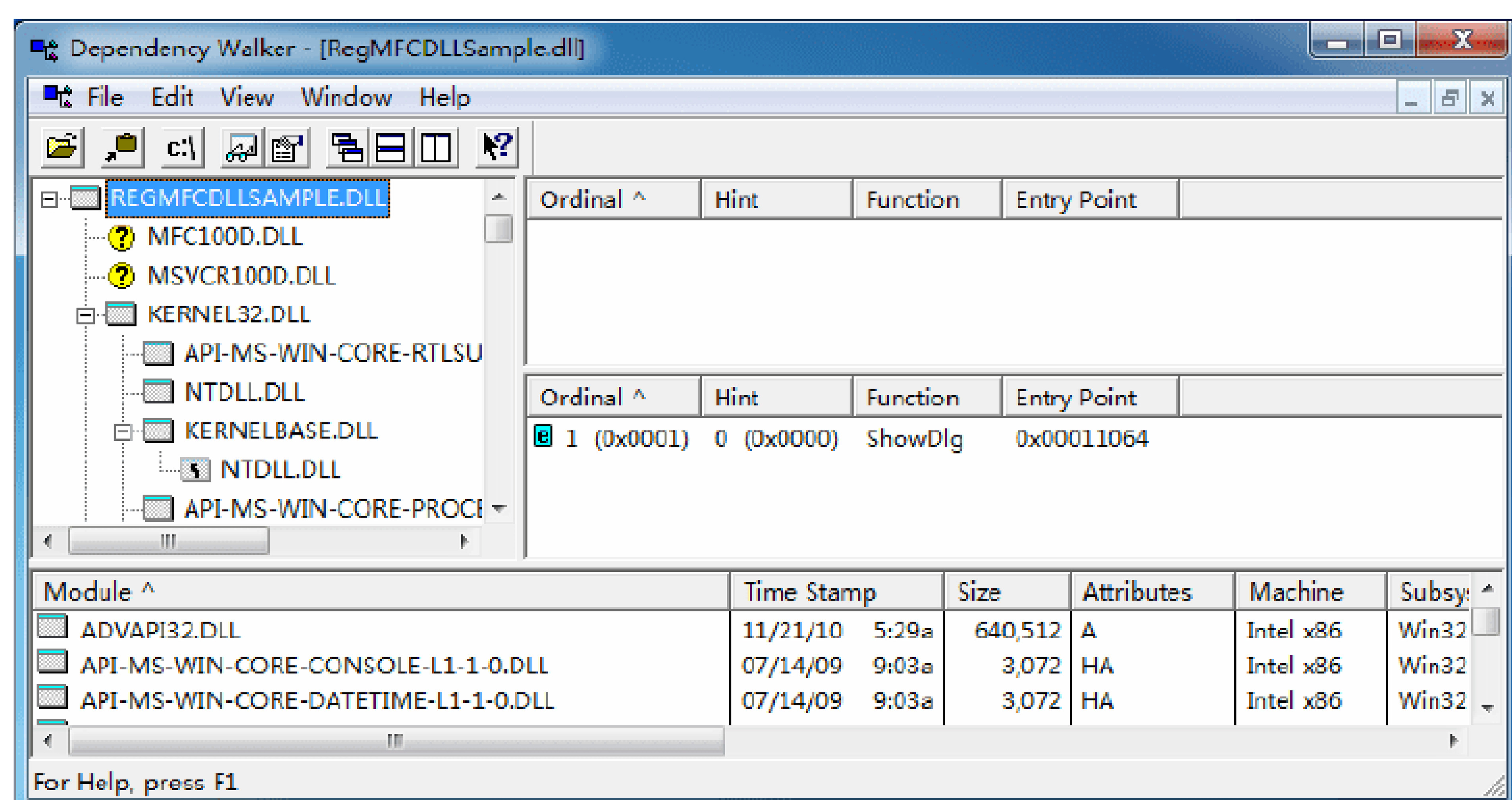


图 22-12　Depends 工具运行界面

在图 22-12 中，显示了在 22.3 节中创建的 DLL 的接口。其中左边树形视图，显示了 DLL 调用的所有其他 DLL 的列表。右边列表部分显示了查看的 DLL 提供的接口函数。底部的列表视图显示了调用的 DLL 模块的信息。在左边树形视图中，选择相应的 DLL，在右边的列表部分，会显示对应的接口函数的列表。其中，Ordinal 列表示函数在 DLL 中的序号或名称，Hint 列表示接口函数在 DLL 内部的序号值，Function 列表示函数名称，Entry Point 列表示函数入口点。读者可以使用此工具查看指定 DLL 所调用的其他 DLL，以及指定 DLL 提供的接口函数。

22.5.2　调试 DLL

在编写程序时，一定会遇到需要调试的情况。DLL 的调试与 EXE 调试方法是类似的，但是在 EXE 调试时，可以直接在要调试的代码行上加上断点进行调试。当 EXE 调用 DLL 时，使用静态链接的方法调用 DLL 接口，则可以在 DLL 需要调试的源代码处设置断点，直接运行 EXE 程序，则程序运行到断点时，会中断以进行调试。但是如果动态加载，断点调试就比较困难。

直接运行 DLL 工程，则会弹出 Executable For Debug Session 对话框。在 Executable file name 文本框中选择要运行用于调试 DLL 的可执行文件，如图 22-13 所示。

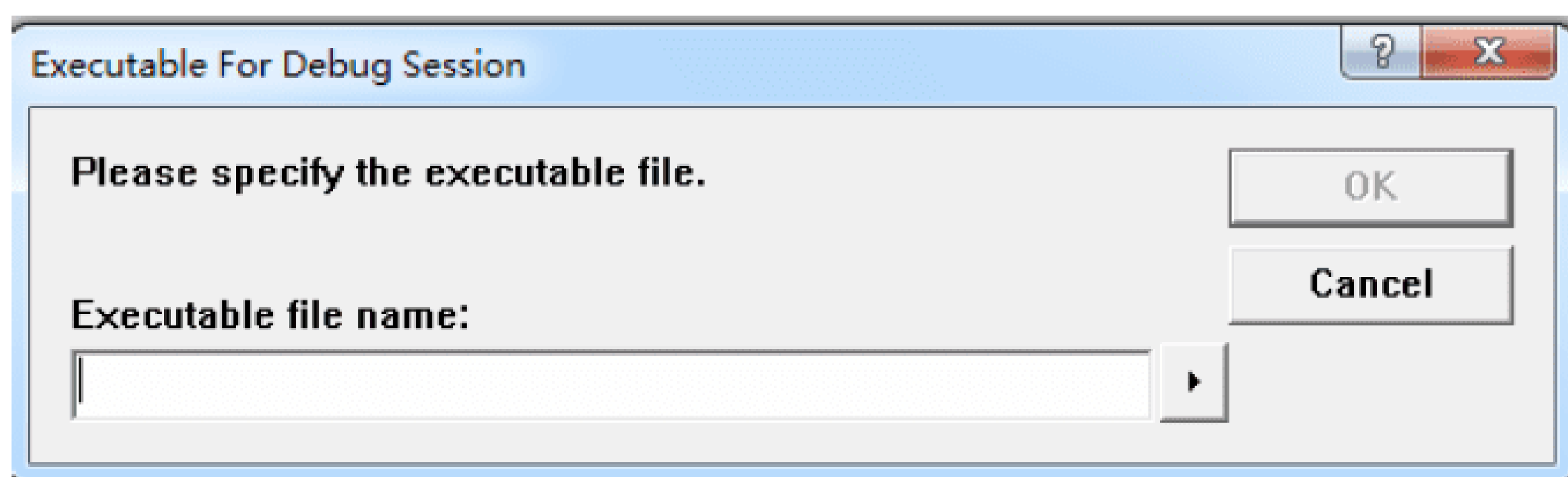


图 22-13　运行设置 DLL 调试的宿主程序

设置好了宿主程序，在 DLL 源代码中要调试的代码行上加入断点，并运行宿主程序，则程序运行到 DLL 的断点处会中断等待进行调试。

除了添加断点进行调试的方法外，其他的调试方法在 DLL 中与在 EXE 中进行调试的方法是一样的。熟练掌握程序调试的方法，可以提高程序的开发效率，并且可以提高代码质量。

22.6　本 章 小 结

本章介绍了在 Visual Studio 2010 环境下进行 DLL 开发的知识和过程。本章重点介绍了非 MFC 动态链接库的应用和 MFC 动态链接库的开发以及 DLL 工具。本章难点在于掌握动态链接库的核心机制，从而深入理解 DLL 的各种应用。第 23 章将介绍有关多线程程序的开发知识。

22.7　习　　题

按照以下提示操作过程，创建一个 DLL 程序，用于实现简单的加法运算。

（1）使用向导创建一个空的 Win32 DLL 项目，向导的设置如图 22-14 所示，在项目中编写如下函数：

```
int addFunction(int a,int b)
{
    return a + b;
}
```

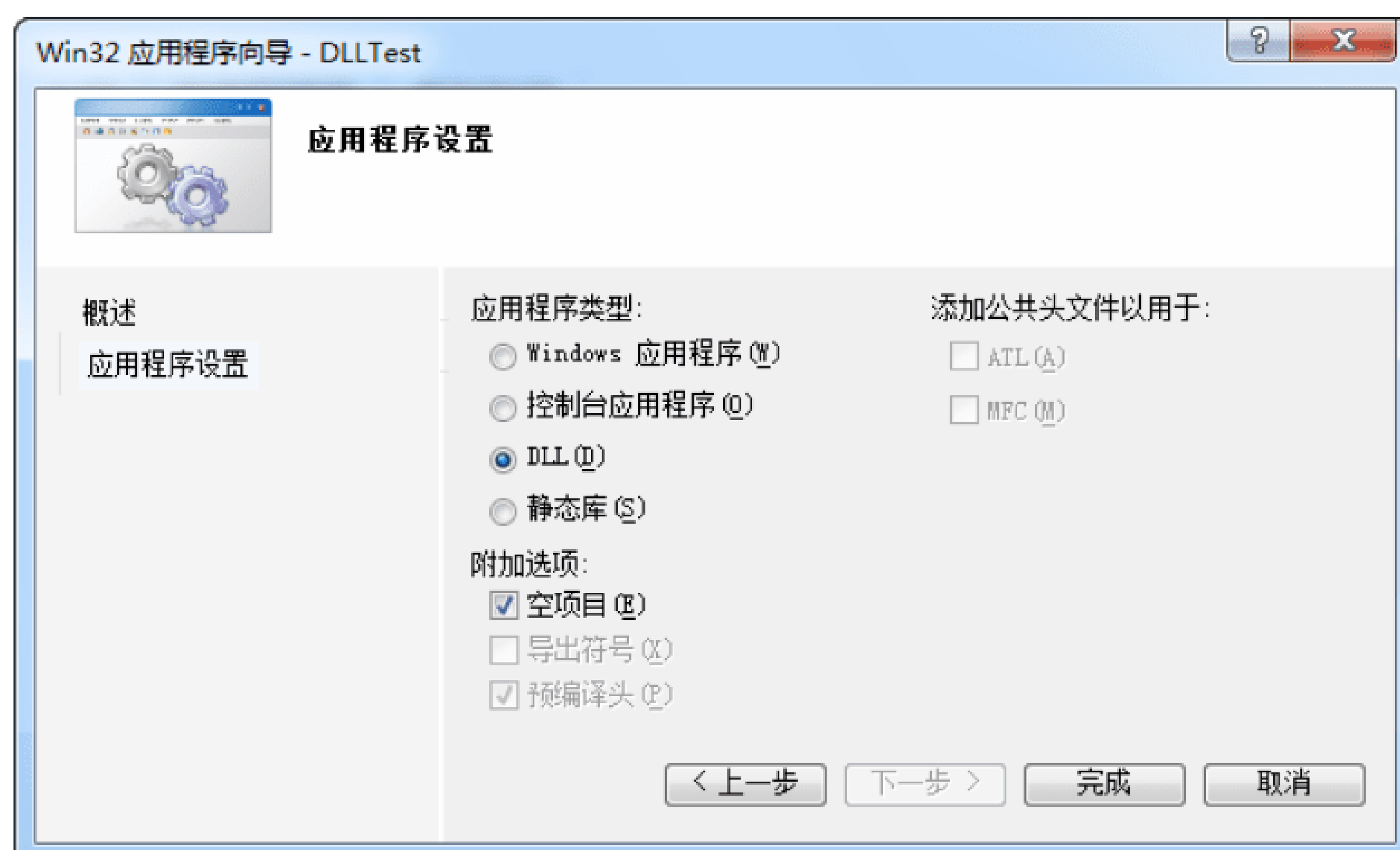


图 22-14　Win32 应用程序向导设置

（2）编译此 DLL 项目，函数 addFunction()要可以被外部程序调用。

（3）创建对话框项目，并设置如图 22-15 所示的界面。

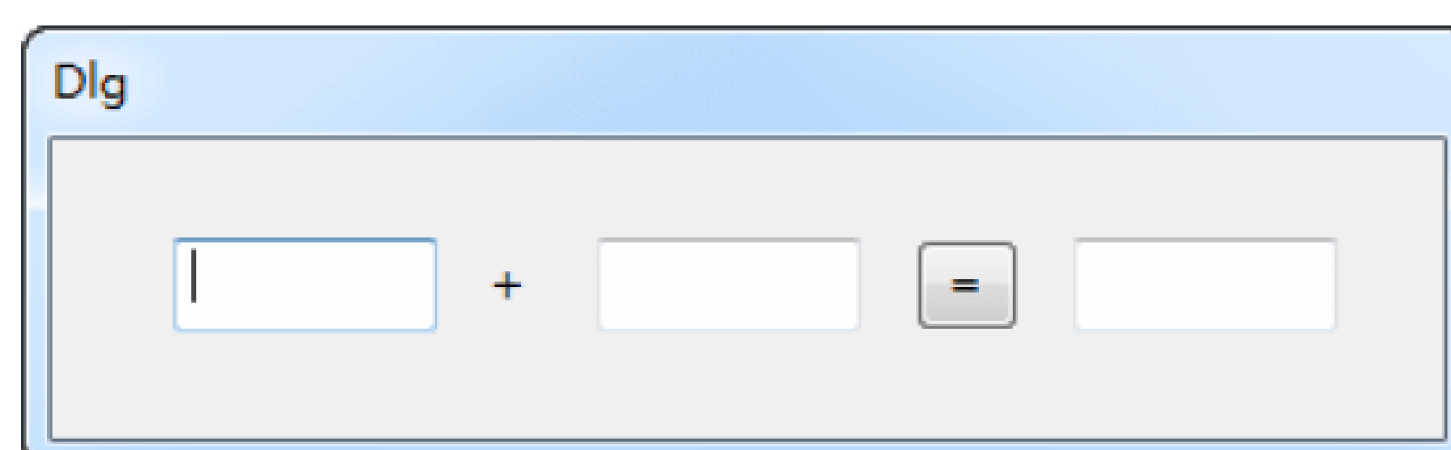


图 22-15　对话框界面

（4）编程实现：单击按钮“=”后会调用（2）所创建的 DLL 中的函数 addFunction()，计算按钮左面两个文本框中的数据的和，并将结果显示在右面的文本框中。

【思路】参考 22.2.1 小节和 22.2.2 小节所讲的内容。

第 23 章　多线程编程

本章讲述 Windows 编程中的多线程程序的开发。在介绍了多线程的引入后，介绍了多线程编程的基础知识和开发方法，然后着重讲解了线程间的通信和线程的同步，最后讲解了一个多线程程序的实例。

23.1　引入多线程

前面章节中讲解的例子都是单线程程序，即一个程序只有一个线程。虽然有些单线程程序可以满足用户的需求，但是随着用户需求越来越复杂，技术更新越来越快，就引入了多线程程序。本节将介绍单线程程序的不足和如何解决单线程程序问题的方法。

23.1.1　单线程的不足

Windows 操作系统是多任务操作系统，可以同时运行多个任务（应用程序）。当每个应用程序运行时，会启动一个主线程，在主线程中用户可以完成程序功能。对于简单的数据计算和一般的数据处理，采用单线程程序是可以处理的。但是用户的需求随着计算机技术的发展也是越来越复杂。

假定这样一个实例，现在需要编写一个银行中间件，用于处理与保险公司之间的结算业务。这个程序需要处理以下几项内容。

- 接收来自保险公司的数据请求，这项任务需要与保险公司端的程序进行数据通信。当然通信方式可能是 TCP/IP，也可能会有专用的应用协议。
- 接收到数据请求后，需要将接收到的数据进行解码，包括校验的处理、错误处理和协议数据解析等。
- 分析完协议数据包后，需要根据协议数据，对接收到的数据按照与银行中心系统之间的通信协议进行编码。
- 将编码后的数据发送到银行中心系统以进行相应的数据操作。
- 接收来自银行中心系统的反馈信息，将反馈信息解码，并返回给保险公司端的程序。

总结上面这个实例，这个银行中间件主要做以下 3 方面的工作。

- 与保险公司端程序之间的数据通信，包括发送和接收。
- 对保险公司端和银行中心系统发送过来的数据进行解码，并对发送给保险公司端和银行中心系统的数据进行编码。
- 与银行中心系统之间的数据通信，包括发送和接收。

在这个实例中，使用单线程编程是无法实现的。单线程只有一个主线程，无论设计时

将哪项工作作为主线程的运行依据都无法完成。如以与保险公司端程序之间的数据通信为主要运行依据，当接收到来自保险公司端程序的数据时，调用数据解析。然后根据数据内容进行数据编码，并将编码后的数据发送给银行中心系统，等待中心系统的反馈。这样处理，看似没有问题，但是，在没有处理完这笔业务时，保险公司端又发送新的数据，怎么办呢？这时就会出现丢失业务处理的情况，或程序响应慢的问题。由此可见，仅仅使用单线程进行编程是无法满足复杂多变的情况的。

23.1.2　解决的问题

从 23.1.1 小节的例子中看到了单线程程序的不足，要解决这个问题，就需要引入了多线程技术。简单地理解，多线程技术就是在一个程序中“同时”运行多个线程。这里之所以将“同时”上加上引号，是因为只有在多核处理器的计算机上，才可以实现真正的同时运行多个线程。在单核处理器计算机中，只是将 CPU 时间分割成多个小的时间片，看上去像是在同时运行。使用多线程可以按以下方法解决 23.1.1 小节的问题。

- 启动负责与保险公司端程序进行数据通信的两个线程，一个用于接收数据，一个用于发送数据，并创建两个缓冲区（也可以是链表），一个缓冲区用于存储接收到的数据，一个用于存储等待发送的数据。
- 启动负责解析从保险公司端程序发送来的数据的线程和负责编码要发送给保险公司端程序的数据的线程。
- 启动负责解析从银行中心系统发送来的数据的线程和负责编码要发送给银行中心系统的数据的线程。
- 启动负责与银行中心系统进行数据通信的两个线程，一个用于接收数据，一个用于发送数据，并创建两个缓冲区（也可以是链表），一个缓冲区用于存储接收到的数据，一个用于存储等待发送的数据。

从上面的分析中可以看出，在此实例中需要启动 8 个线程，这样可以使得数据接收和数据处理的过程不会受到阻碍。这也是引入多线程的原因。从 23.2 节开始，就开始介绍如何开发多线程程序。

23.2　进程和线程

Windows 平台和 Visual Studio 2010 环境都为多线程的编程提供了支持。本节介绍有关多线程编程的基础知识。首先介绍进程和线程的概念，然后介绍多线程工具 Spy++，最后分别介绍 Win32 和 MFC 对多线程编程的支持。

23.2.1　Spy++

Spy++（SpyXX.EXE）是一个基于 Win32 的工具，可以使用图形的方式查看系统的进程、线程、对话框和对话框消息。Spy++具有工具栏和超链接，使得操作更方便。提供刷新命令更新当前激活的视图以及对话框查找工具，并且还提供字体对话框设置显示的字体。

另外，Spy++还可以保存和恢复用户的设置。

Spy++工具只支持同一时间运行单个实例，即当用户在打开 Spy++工具后再次启动，则会将已经打开的 Spy++工具移到最上端。它是一个只读工具，使用 Spy++不能修改程序的操作。使用 Spy++工具的步骤如下。

（1）选择“工具”| Spy++命令，打开 Spy++对话框，其中包含系统中的所有窗体。单击任何一个选项，弹出“属性检查器”对话框。其中显示了选择的窗体的信息，包括样式、对话框、类和对应的进程信息，如图 23-1 所示。

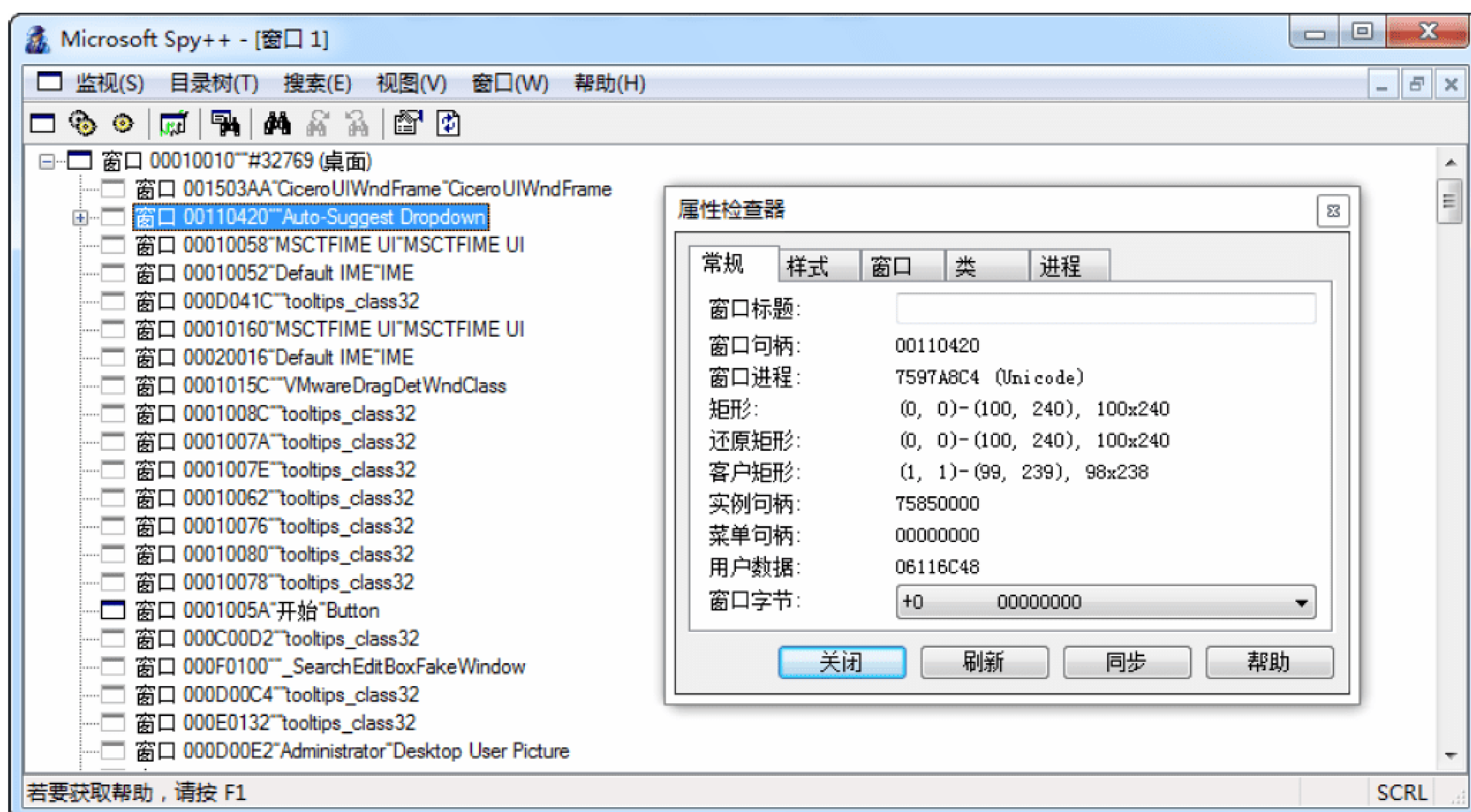


图 23-1　Spy++工具界面

（2）选择“监视”|“进程”命令，打开“进程 1”对话框，以树形结构显示了当前系统包含的进程。展开任何一项，会在下层显示出此进程包含的线程。右击其中的进程，弹出“进程属性”对话框。其中显示了选择的进程信息，如图 23-2 所示。

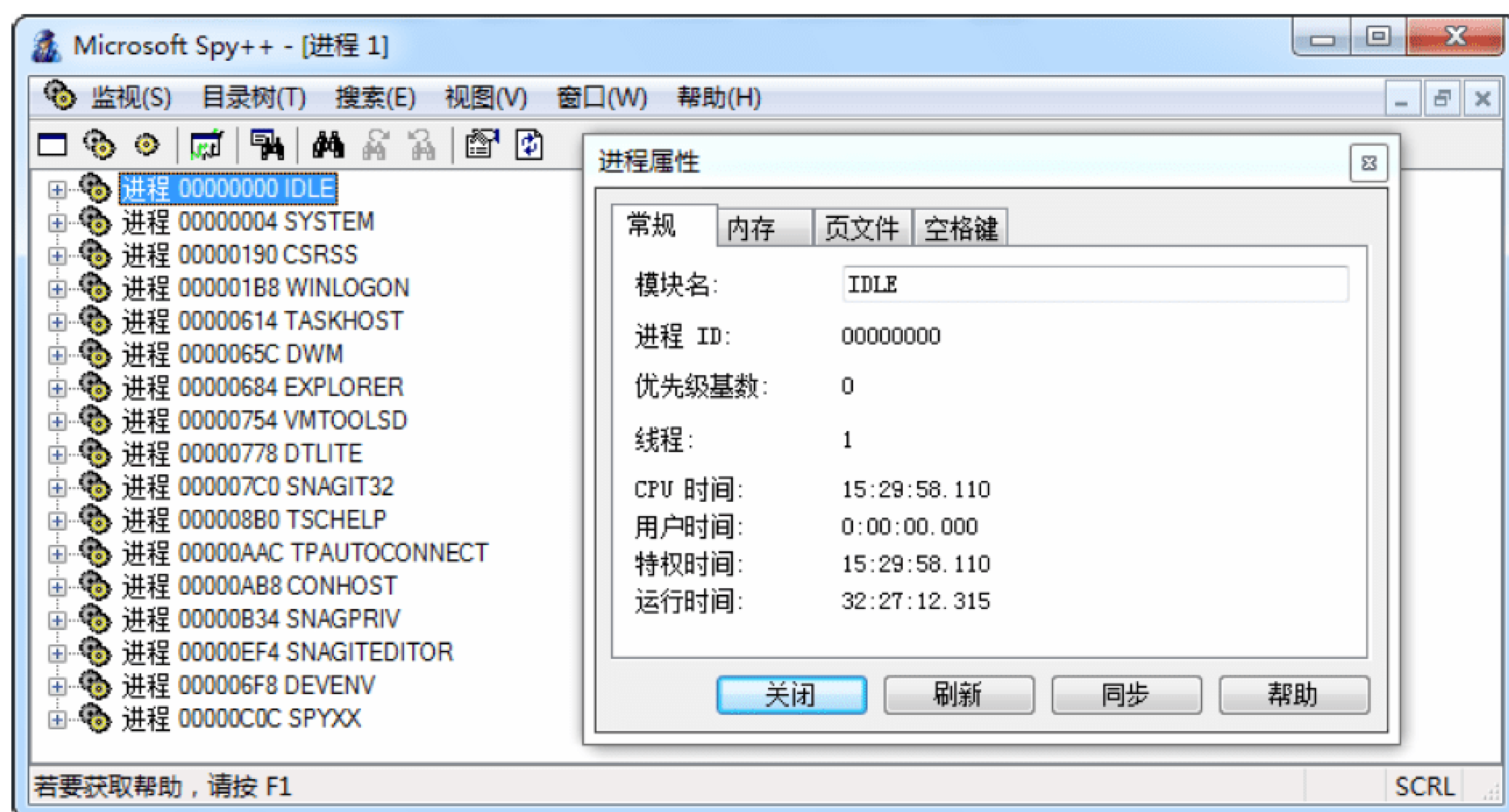


图 23-2　Spy++进程界面

（3）选择“监视”|“线程”命令，打开“线程 1”对话框，以树形结构显示了当前系统包含的所有线程。单击其中的线程，弹出“线程属性”对话框。其中显示了选择的线程信息，如图 23-3 所示。

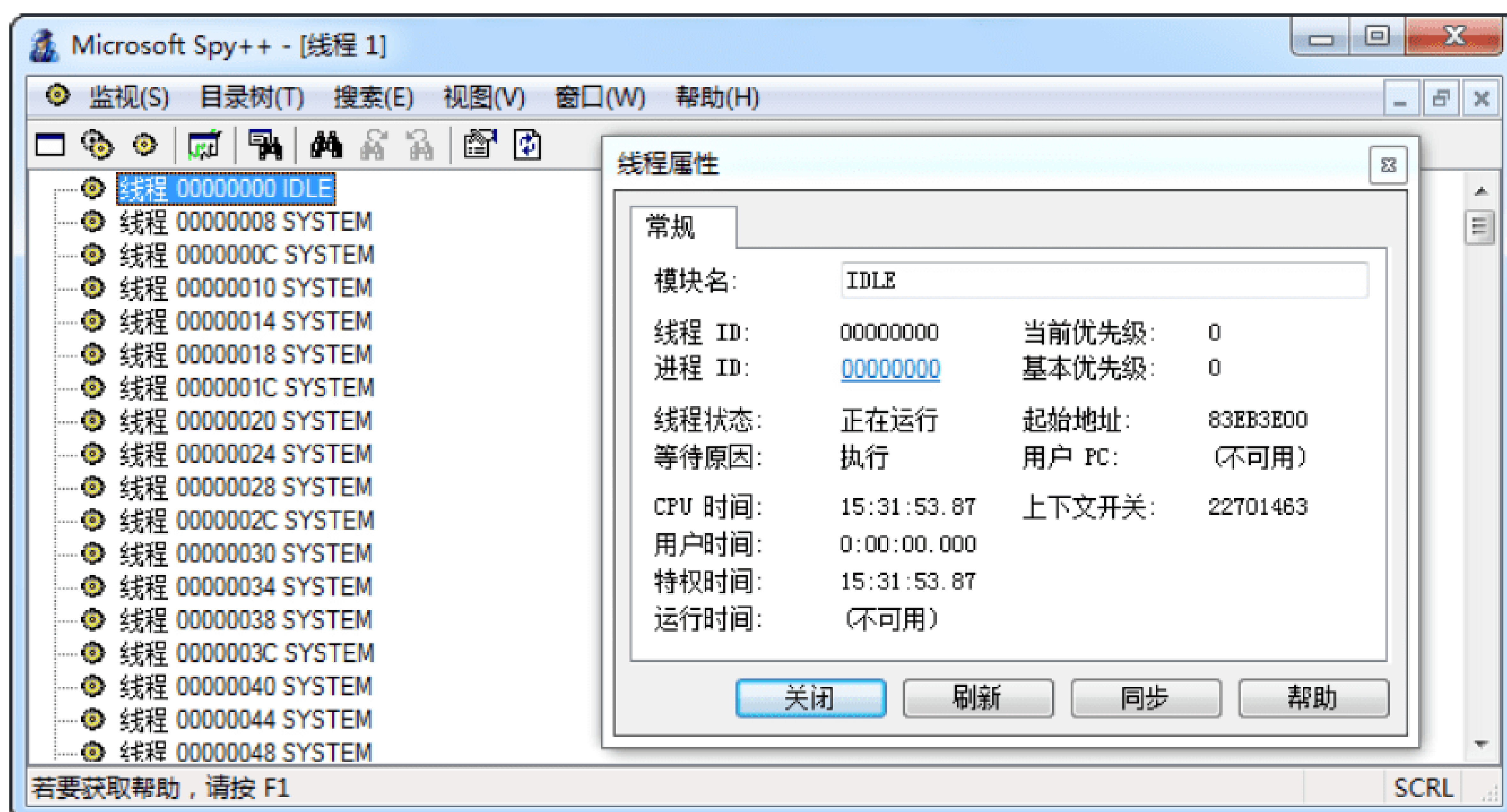


图 23-3　Spy++线程界面

（4）在对话框列表中，选择要追踪消息的对话框后，选择“监视”|“日志消息”命令，打开“消息选项”对话框，选择“消息”选项卡，如图 23-4 所示。选择要追踪的消息类型，单击“确定”按钮后，程序开始追踪指定对话框的指定消息。也可以在“输出”选项卡中指定追踪结果的保存方式，如存入文件等。追踪完成后，可以选择“消息”|“停止记录”命令，停止消息追踪。

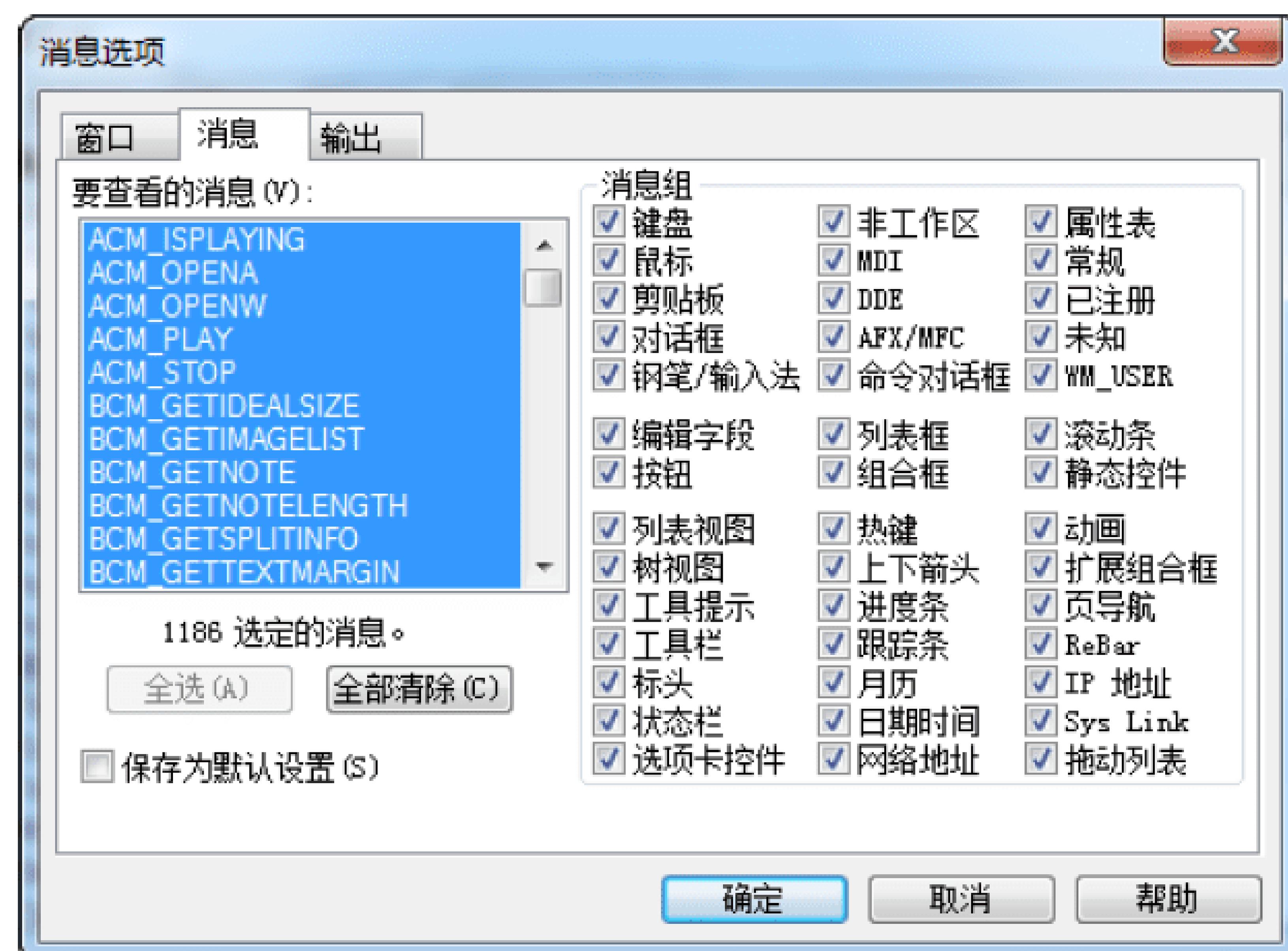


图 23-4　Spy++工具中“日志消息”选项设置

23.2.2　多线程 Win32 API

VC 使用 Win32 API 可以创建多线程应用程序。如可以同时处理键盘和鼠标的输入，第一个线程处理键盘输入，第二个线程过滤鼠标动作，而第三个线程可以根据鼠标和键盘

线程接收到的数据更新界面显示。同时，其他线程可以访问磁盘文件或从通信端口处获取数据。在 VC 中，有两种方式可以编写多线程程序：使用 MFC 类库或使用 C 运行库和 Win32 API。Win32 API 通过下面的函数提供对多线程编程的支持。

- CreateThread()函数用于创建新线程。
- ExitThread()函数用于结束线程。
- GetThreadPriority()函数用于获得线程优先级。
- SetThreadPriority()函数用于设置线程优先级。
- SuspendThread()函数用于挂起线程。
- ResumeThread()函数用于恢复线程。

使用 Win32API 开发多线程程序的过程，将在 23.3.1 小节中介绍。

23.2.3　MFC 对多线程编程的支持

MFC 中的 CWinThread 对象提供对多线程编程的支持。大部分情况下，不需要显式地创建 CWinThread 对象，而是调用 AfxBeginThread()函数创建 CWinThread 对象。MFC 分为两种类型的线程：用户界面线程和工作者线程。用户界面线程通常用于处理用户输入，响应用户发送的事件和消息。工作者线程通常用于完成任务，如不需要输入的计算。Win32 API 不区分这两种线程，只需要知道线程的启动地址就可以启动线程。

MFC 在用户界面中提供消息处理事件。CWinApp 是一个用户界面线程对象，派生自 CWinThread，处理用户产生的事件和消息。关于 CWinApp 类的使用，将在 23.3 节进行介绍。

23.3　开发多线程程序

有了前面对多线程编程的基础知识，就可以进行多线程程序开发了。本节介绍多线程程序的开发方法。包括使用 Win32 API 函数开发多线程程序的方法、使用 MFC 创建两种线程的方法、如何挂起线程和终止线程、如何使线程处于睡眠状态，最后讲解了几个多线程示例。

23.3.1　使用 Win32 API 函数开发

CreateThread()函数用来创建一个新线程。创建线程必须指定新线程要执行的开始地址。通常，开始地址是在程序代码中定义的函数名。函数需要一个参数和返回一个 DWORD 值。一个进程可以并发多个线程执行相同的函数。下面的代码显示了如何创建一个新线程执行本地定义的 ThreadFunc()函数。

```
01  DWORD WINAPI ThreadFunc( LPVOID lpParam )        //线程处理函数
02  {
03      printf("启动线程\n");                        //输出提示信息
04      return 0;                                    //函数返回
05  }
06  int main(int argc, char* argv[])                 //程序主函数
```

```
07  {
08      DWORD dwThreadId;                                   //线程 ID
09      HANDLE hThread;                                     //线程句柄
10      //创建线程
11      hThread = CreateThread(NULL,0,
12          (LPTHREAD_START_ROUTINE)ThreadFunc,NULL,0,&dwThreadId);
13      if (hThread == NULL)
14          printf("创建线程失败");                          //输出错误信息
15      Sleep(2000);                                        //延时
16      CloseHandle( hThread );                             //关闭句柄
17      printf("主线程结束\n");                              //输出提示信息
18      return 0;
19  }
```

在上面的代码中，main()是主函数，程序调用 CreateThread()函数创建一个新线程，新线程执行 ThreadFunc()函数。为了简便，示例没有向线程传递参数，但是参数指针可以指向任何类型的数据或结构，或者传入一个 NULL 指针。ThreadFunc()函数只是简单地将提示信息显示在屏幕上。而主线程会根据 CreateThread()函数的返回值判断创建线程是否成功，调用 Sleep()函数使得主线程挂起 2000 毫秒，最后关闭线程句柄，在屏幕上提示后退出。为什么要调用 Sleep()函数，会在后面介绍（此处是一个重要概念）。程序运行的效果如图 23-5 所示。

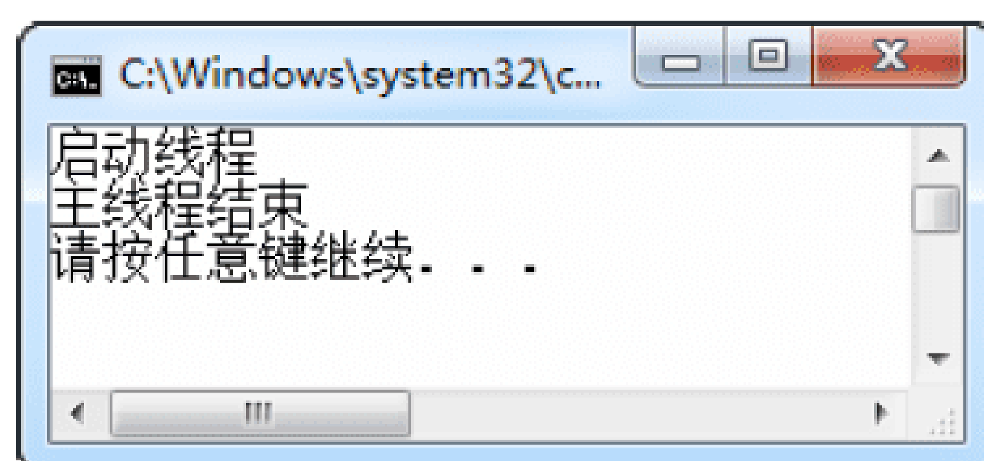


图 23-5　Win32 API 函数开发的多线程程序运行效果

23.3.2　MFC 用户界面线程的开发

用户界面线程独立于应用程序其他的可执行线程，处理用户输入和响应用户事件。主应用线程由 CWinApp 派生类创建和启动，因此，这里讲述创建单独的用户界面线程的方法。第一步要创建派生自 CWinThread 的类，并使用 DECLARE_DYNCREATE 和 IMPLEMENT_DYNCREATE 宏声明和实现此类。这个类必须重载一些函数，或根据需要重载其他部分函数，如表 23-1 所示。

表 23-1　创建界面线程时重载的函数

函　数　名	功　　能
ExitInstance()	当线程终止时，执行析构功能。通常需要重载
InitInstance()	完成线程实例的初始化。必须重载
OnIdle()	完成线程空闲时的工作。一般不需要重载
PreTranslateMessage()	在分配消息给 TranslateMessage()函数和 DispatchMessage()函数前过滤消息。一般不需要重载
ProcessWndProcException()	处理线程消息和命令句柄发送的未处理的异常。一般不需要重载
Run()	线程的运行函数。几乎不需要重载

MFC 通过参数重载提供两个版本的 AfxBeginThread()函数，一个用在用户界面线程中，另一个用在工作线程中。用于创建用户界面线程的 AfxBeginThread()函数原型为：

```
CWinThread* AfxBeginThread(
  CRuntimeClass* pThreadClass,//派生自 CWinThread 类的 RUNTIME_CLASS
  int nPriority = THREAD PRIORITY NORMAL,
                                //指定线程的优先级级别，此参数默认值为默认级别
  UINT nStackSize = 0,          //指定线程的堆栈大小，默认值与创建线程的堆栈大小相同
  DWORD dwCreateFlags = 0,      //表示线程创建时的状态
  LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );//表示线程的安全属性
```

AfxBeginThread()函数为界面线程提供了大部分的实现代码。创建指定类的对象，使用用户提供的信息初始化，并且调用 CWinThread::CreateThread()函数启动可执行线程。

23.3.3　MFC 工作者线程的开发

工作者线程通常用于处理后台任务，即用户不需要等待其完成才能继续进行的任务。如重新核算任务和后台打印任务都可以作为工作者线程。创建工作者线程相对比较简单，分为两步：完成控制函数和启动线程。一般情况下，不需要从 CWinThread 中派生类。如果需要，可以创建特殊版本的 CWinThread，但是对于大部分简单的工作者线程是不需要从 CWinThread 中派生类的，也就是可以不做任何修改地使用 CWinThread。用于创建工作者线程的 AfxBeginThread()函数原型为：

```
CwinThread* AfxBeginThread(
  AFX_THREADPROC pfnThreadProc,//指定控制函数的地址，用于指向执行线程运行函数
  LPVOID pParam,
  int nPriority = THREAD_PRIORITY_NORMAL,   //指定线程的优先级级别，此参数默认
                                            //值为默认级别
  UINT nStackSize = 0,                      //指定线程的堆栈大小
  DWORD dwCreateFlags = 0,                  //线程创建时的状态
  LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );//表示线程的安全属性
```

AfxBeginThread()函数创建和初始化 CWinThread 对象，启动并返回地址，用户可以在后面引用此对象。控制函数用于定义线程。当进入此函数时，线程启动；当退出此函数时，线程终止。控制函数的原型如下：

```
UINT MyThreadFunction( LPVOID pParam );
```

其中，pParam 参数是一个 32 位的值，表示当创建线程对象时，传给构造函数的参数值。控制函数可以按照约定的方式解析此参数值，可以作为精度值，也可以是指向包含多个参数的结构的指针，或者是忽略此参数。如果参数指向结构，则调用者不仅可以通过结构将值传给函数，还可以从线程返回数据给线程。当函数终止时，返回一个 UINT 值表示终止的原因。通常，使用 0 退出码表示成功，其他值表示发生错误的代码。下面代码首先定义了 Cmaths 类，该类支持动态创建。

```
01  class Cmaths  : public CObject                //Cmaths 类定义
02  {
03      DECLARE_DYNAMIC( Cmaths )
04      public:
05      Cmaths();
06  };
```

```
07  IMPLEMENT_DYNAMIC( Cmaths, Cobject )
08  Cmaths::Cmaths()
09  {
10  }
```

从上面代码中可以看出，要使得派生类支持动态创建，则需要在头文件中加上 DECLARE_DYNAMIC 宏定义，在源文件中加上 IMPLEMENT_DYNAMIC 宏定义，否则是无法在下面的代码中使用 IsKindOf()函数的。下面是工作者线程控制函数的代码。

```
01  UINT MyThreadProc( LPVOID pParam )      //线程处理函数
02  {
03      Cmaths* pObject = (Cmaths*)pParam;  //定义 CMaths 对象
04      if ((pObject == NULL) ||
05          (!pObject->IsKindOf(RUNTIME CLASS(Cmaths))))
06      {                                   //判断是否为指定对象
07          AfxMessageBox("参数传入失败。Prepare Exit Thread");
08          return 1;
09      }
10      while (true)                        //执行 while 循环，进行计数
11      {
12          globalCounter++;
13          if (globalCounter > 99999999)
14              globalCounter=0;
15          Sleep(1000);
16      }
17      return 0;
18  }
```

下面是使用 AfxBeginThread()函数创建线程的代码。

```
01  void CmultiThreadFuncSampleDlg::OnButtonStartThread()
02  {
03      Cmaths* pMathObject = new Cmaths();      //创建对象
04      pThread = new CwinThread();              //创建线程
05      pThread->m_bAutoDelete = false;          //设置是否自动删除为 false
06      pThread = AfxBeginThread(MyThreadProc, pMathObject);//启动线程
07      if (pThread != NULL)
08          MessageBox("启动线程成功");              //输出提示信息
09  }
```

上面的例子启动了一个工作者线程，并传入 Cmaths 类的对象。在工作者线程的函数中，判断传入的指针是否指向 Cmaths 类的对象。如果是，则提示用户，并返回 0，表示线程控制函数正确返回；如果不是，则提示用户传入的类不正确，并返回 1，表示线程控制函数在执行过程中发生错误。程序运行的效果如图 23-6 所示。

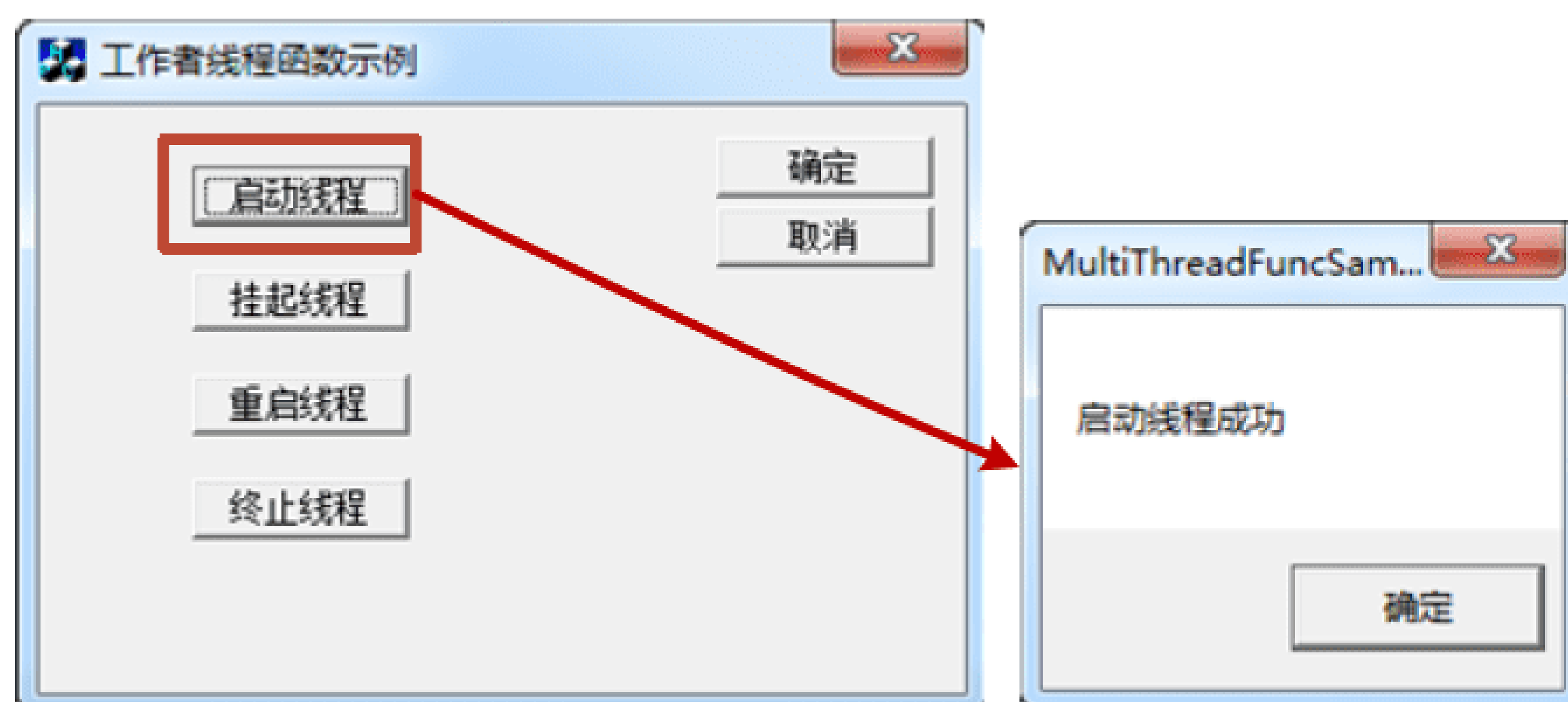


图 23-6　启动 MFC 工作者线程运行效果

23.3.4　挂起线程

使用 SuspendThread()函数可以挂起指定的线程，挂起线程也就是暂停线程，使得线程不执行应用程序代码。每个线程都具有一个挂起计数，不能大于 MAXIMUM_SUSPEND_COUNT。此挂起计数大于 0，表示线程被挂起，否则，线程没有被挂起，正常运行。其函数原型为：

```
DWORD SuspendThread( HANDLE hThread );
CWinThread:: SuspendThread  DWORD SuspendThread ( );
```

其中，hThread 参数表示要挂起的线程的句柄。如果函数成功，则返回值为线程以前挂起的次数，否则，返回值为 0xFFFFFFFF。要获取错误原因，可以调用 GetLastError()函数。如果函数成功，则指定的线程的运行会挂起，并且会增加线程的挂起计数。

要恢复线程的运行，可以使用 ResumeThread()函数减少挂起线程的挂起次数。当线程的挂起次数减到 0 时，线程会恢复执行。其函数原型如下：

```
DWORD ResumeThread( HANDLE hThread );
CWinThread:: ResumeThread  DWORD ResumeThread( );
```

其中，hThread 参数表示要重新启动的线程的句柄。如果函数成功，则返回值为线程以前挂起的次数，否则，返回值为 0xFFFFFFFF。要获取错误原因，可以调用 GetLastError()函数。也就是说，如果返回值为 0，则指定线程原来没有被挂起；如果返回值为 1，表示指定的线程原来被挂起，但是现在重新启动了；如果返回值大于 1，则指定的线程仍然处在挂起状态。

从上面的函数原型中可以看出，调用 Win32 API 的线程函数与 CWinThread 类的函数是相同的。因为 CWinThread 是对 Win32 API 线程函数的封装，所以不同之处只在于 CWinThread 类的成员函数中没有传入指定线程句柄的参数。由于 CWinThread 类中保存了 m_hThread 成员函数用于保存线程句柄，因此这两种方式的调用作用是相同的。下面显示了挂起线程和重新启动线程的代码。

```
01  void CmultiThreadFuncSampleDlg::OnButtonSuspendThread()
02  {
03      //挂起线程
04      pThread->SuspendThread();
05      CString info;
06      info.Format("挂起后 globalCounter=%d", globalCounter);
07      MessageBox(info, "提示");
08  }
09  void CmultiThreadFuncSampleDlg::OnButtonResumeThread()
10  {
11      //重新启动线程
12      CString info;
13      info.Format("重新启动前 globalCounter=%d", globalCounter);
14      MessageBox(info, "提示");
15      pThread->ResumeThread();
16  }
```

在上面的代码中，OnButtonSuspendThread()函数挂起线程。在挂起线程后，将 globalCounter 变量的值显示出来。OnButtonResumeThread()函数重新启动线程，在重新启

动前，显示出 globalCounter 变量的值。从中可以看出，在挂起线程后，线程并没有工作，即 globalCounter 值没有发生变化。程序运行的效果如图 23-7 所示。



图 23-7　挂起和恢复线程运行效果

23.3.5　终止线程

终止线程有两种情况：控制函数退出或提前终止线程。如文字处理程序使用后台打印程序线程，如果打印成功完成，则后台打印线程的控制函数可以正常退出；而如果用户想要取消打印，则后台打印线程则不得不永久地终止。本小节介绍这两种终止线程的方法和如何获取线程终止的退出代码。

对于工作者线程来说，正常终止线程非常简单，退出控制函数，并返回终止原因的代码值。用户可以使用 AfxEndThread()函数或 return 语句。通常，返回 0 表示成功完成线程处理。

对于用户界面线程来说，过程是相同的，在用户界面线程中，调用 PostQuitMessage()函数。其中唯一的一个参数就是线程退出码，与工作者线程一样，0 表示线程成功完成。

提前终止线程的过程也很简单，在线程中调用 AfxEndThread()函数。唯一的参数表示线程退出原因码，终止线程的执行，释放线程的堆栈，卸载所有加载到线程中的 DLL，并从内存中删除线程对象。此函数的原型为：

```
void AfxEndThread( UINT nExitCode );
```

其中，nExitCode 参数表示线程的退出原因码。AfxEndThread()函数必须在要终止的线程中调用，如果要在一个线程中调用此函数终止另一个线程的运行，则需要在两个线程之间建立通信方式。

使用::GetExitCodeThread()函数可以获取线程的退出码。其函数原型为：

```
BOOL GetExitCodeThread( HANDLE hThread, LPDWORD lpExitCode );
```

其中，hThread 参数是要获取退出码的线程的句柄，在 CWinThread 对象中，m_hThread 数据成员中存放此句柄。lpExitCode 参数用于存放获取的线程终止码，是一个 32 位的整型值。如果函数操作成功，则返回非 0（true）；否则返回 0（false）。要获取错误原因，可以调用 GetLastError()函数。如果查询的线程还没有退出，则返回的终止码为 STILL_ACTIVE。如果线程已经终止，则返回值如下。

- ❑ 在 ExitThread()函数或 TerminateThread()函数中指定的退出码。
- ❑ 从线程控制函数中返回的值。

- ❑ 线程所属的进程的退出码。

在 MFC 中，要获取 CWinThread 对象的退出码，需要做特殊处理。默认情况下，当 CWinThread 线程终止时，系统会删除线程对象，因为此对象不再存在，所以不能访问 m_hThread 数据成员。要避免这种情况，需要做下面两步的处理。

- ❑ 设置 m_bAutoDelete 数据成员的值为 false，这使得 CWinThread 对象在终止线程后不会自动删除对象，因此当线程终止后，仍然可以访问 m_hThread 数据成员。使用此种方式，用户必须处理 CWinThread 对象的销毁工作。
- ❑ 单独存储线程的句柄。线程创建后，使用::DuplicateHandle()函数复制 m_hThread 数据成员到变量中，并通过此变量访问此句柄。使用这种方式，对象被自动删除后，用户仍然可以查找到线程终止的原因。要注意在复制句柄前不能终止线程。比较好的方式是在使用 AfxBeginThread()函数创建线程时，传入 CREATE_SUSPENDED 参数，挂起线程，然后存储句柄，最后通过调用 ResumeThread()函数重新启动线程。

使用这两种方式都可以判断线程的退出代码。下面的代码将 23.3.3 小节中的代码稍做修改，演示了如何获取线程的退出码。

```
01  void CmultiThreadFuncSampleDlg::OnButtonStopThread()//停止线程
02  {
03      bThreadRunning = false;                    //设置线程运行状态为 false
04      DWORD exitCode;
05      CString info;
06      if (GetExitCodeThread(pThread->m_hThread,&exitCode))//获取退出码
07      {
08          info.Format("线程退出码=%u", exitCode);            //输出提示信息
09          MessageBox(info, "提示");
10      }
11      else
12          MessageBox("获取线程退出码时发生错误", "提示");
13  }
```

上面代码通过设置线程工作标识变量 bThreadRunning 的值为 false 终止线程的运行，通过 GetExitCodeThread()函数获取线程的退出码。程序运行的效果如图 23-8 所示。

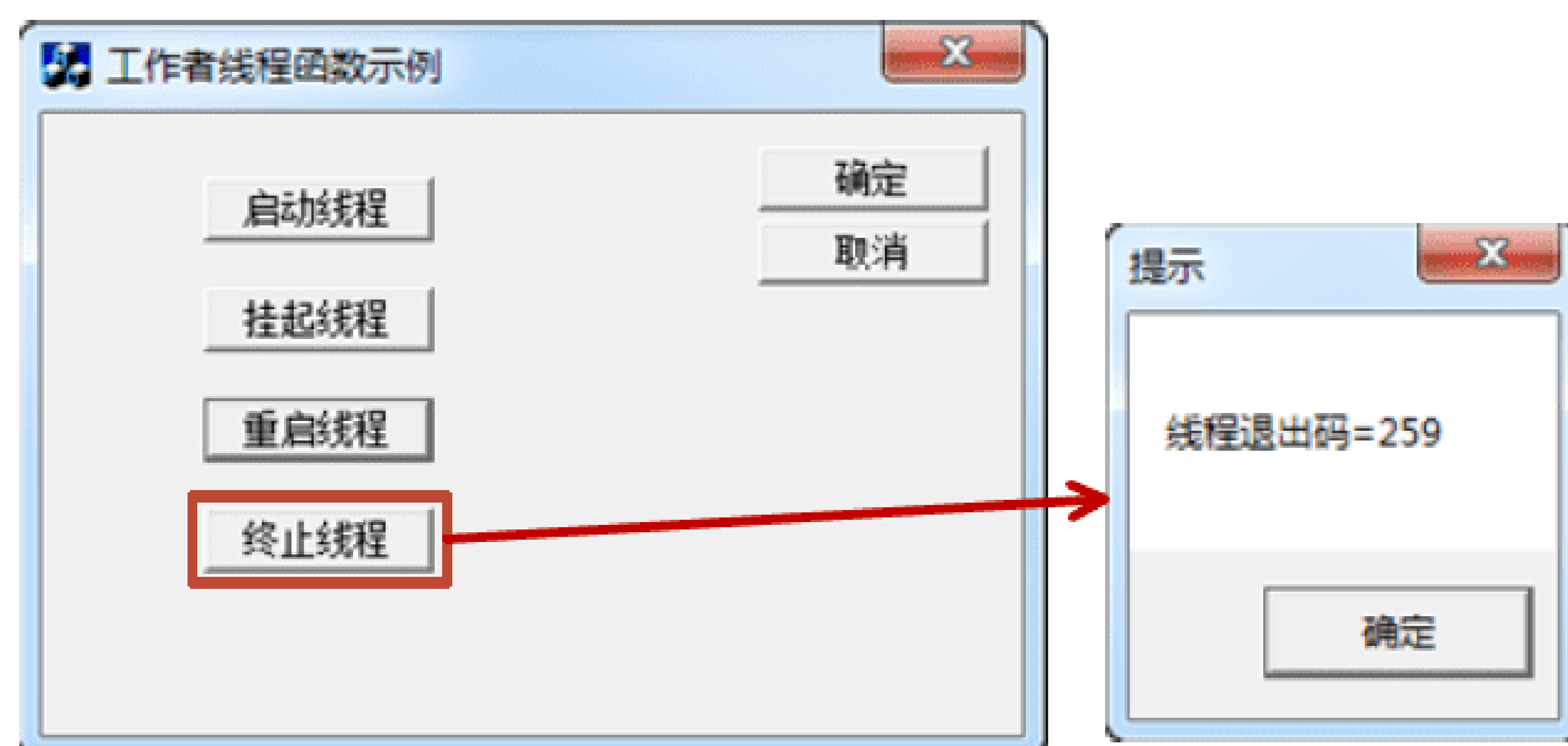


图 23-8　终止线程运行效果

23.3.6　使线程睡眠

因为每个进程或线程都有 CPU 分配的工作时间，如果线程的运行代码是无限循环，则

线程会极力抢占 CPU，使得 CPU 占用率比较高，阻碍了其他进程的运行，进而看上去像系统“死机”。这是程序编写要考虑的基本问题，系统提供了 Sleep()函数，使得线程可以处于睡眠状态一定的时间，进而使 CPU 可以合理分配资源。其函数原型为：

```
VOID Sleep(  DWORD dwMilliseconds  );
```

其中，dwMilliseconds 参数指定线程处于睡眠状态的时间，单位是毫秒。如果传入 INFINITE，则线程会无限地处于睡眠状态。实际上，Sleep()函数就是挂起当前线程一个指定的时间段。此函数没有返回值。在线程中创建对话框时，使用 Sleep()函数要注意，因为 Windows 会将消息发送给所有的对话框，而当处理对话框的线程处于睡眠状态时，则不能接收消息，从而使得程序进入死锁状态。因此，当线程创建对话框时，使用 MsgWaitForMultipleObjects()函数或 MsgWaitForMultipleObjectsEx()函数，而不要使用 Sleep()函数。在 23.3.3 小节中的 MyThreadProc()函数中，就使用了 Sleep()函数，因为此函数中没有创建对话框，所以，这样使用是没有问题的。

23.3.7　启动和关闭记事本

下面连续的 3 个小节开始以操作记事本程序为例，介绍有关线程方面的编程。为了不影响主线程的工作，要启动记事本需要创建一个新线程启动记事本工具，并在线程中对记事本进行控制，如控制记事本关闭。代码如下：

```
01  void CexecNoteSampleDlg::OnButtonStart()    //启动记事本
02  {
03      if (bRunning)
04          return;                              //如果已经运行，则退出
05      pThread = new CwinThread();              //创建线程对象
06      pThread->m_bAutoDelete = false;          //设置自动删除值为 false
07      bRunning = true;                         //设置正在运行值为 true
08      //启动线程
09      pThread = AfxBeginThread(StartAndCloseThreadProc, NULL);
10      if (pThread == NULL)
11          MessageBox("启动记事本失败");
12  }
13  void CexecNoteSampleDlg::OnButtonClose()    //关闭记事本程序
14  {
15      bRunning = false;                        //设置运行值为 false
16  }
```

上面代码中的 OnButtonStart()函数负责启动记事本工作线程，线程入口函数为 StartAndCloseThreadProc()，此函数的定义代码如下：

```
01  int bRunning = 0;
02  UINT StartAndCloseThreadProc( LPVOID pParam )      //线程处理函数
03  {
04      STARTUPINFO si;                                //定义启动信息结构
05      PROCESS_INFORMATION pi;                        //定义进程信息结构
06      ZeroMemory( &si, sizeof(si) );                 //初始化结构变量
07      si.cb = sizeof(si);                            //赋值结构化长度
08      //启动记事本进程
09      if( !CreateProcess(NULL,"notepad.exe",NULL,NULL,false,0,NULL,
10          NULL,&si,&pi))
```

```
11          AfxMessageBox("启动记事本进程失败.");
12      while(bRunning)
13      {
14          if (bRunning == 2)
15              SuspendThread(pi.hThread);                      //挂起
16          else if (bRunning == 3)
17              ResumeThread(pi.hThread);                       //恢复
18          Sleep(1000);
19      }
20      //枚举记事本窗口
21      EnumWindows((WNDENUMPROC)CloseNoteApp, (LPARAM)pi.dwProcessId);
22      if (pi.hProcess)
23          CloseHandle( pi.hProcess );                         //关闭进程
24      if (pi.hThread)
25          CloseHandle( pi.hThread );                          //关闭线程
26      AfxMessageBox("启动的记事本已经关闭");
27      return 0;
28  }
```

上面代码是启动记事本工作的线程函数，其中调用 CreateProcess()函数创建记事本进程，并且通过全局变量 bRunning 控制记事本是否继续工作。当 bRunning 值为 0 时，则会退出 while 循环，执行 EnumWindows()函数。此函数的作用是根据获得的进程 ID 查找与之匹配的进程句柄，也就是查找到刚才打开的记事本程序，并执行对应的操作。此处的操作通过 CloseNoteApp()函数关闭记事本程序，其代码如下：

```
01  BOOL CALLBACK CloseNoteApp(HWND hwnd,LPARAM lParam)//关闭记事本程序
02  {
03      DWORD dwID;
04      GetWindowThreadProcessId(hwnd, &dwID);          //获取记事本进程 ID
05      //关闭记事本
06      if(dwID == (DWORD)lParam)
07          PostMessage(hwnd,WM_CLOSE,0, 0);
08      return true;
09  }
```

上面的代码是 EnumWindows()函数的回调函数，即 CloseNoteApp()函数，每枚举成功一次，也就是每查找到一个对话框后，就会执行此回调函数。函数的 hWnd 参数是查找到的对话框句柄，lParam 参数是通过 EnumWindows()函数传入的要查找的进程 ID 参数值。因此在此函数中，会通过 GetWindowThreadProcessId()函数获得 hWnd 对话框句柄对应的进程 ID，并判断此进程 ID 与传入的参数的进程 ID 是否相等。如果相等，则是找到的对话框，并向其中发送关闭消息 WM_CLOSE。如果此函数返回 true，则 EnumWindows()函数不会继续枚举；如果返回 false，则函数会继续枚举窗体。

在 StartAndCloseThreadProc()线程入口函数中，需要调用 CloseHandle()函数关闭记事本线程和进程，并以对话框的方式提示用户。程序运行的效果如图 23-9 所示。

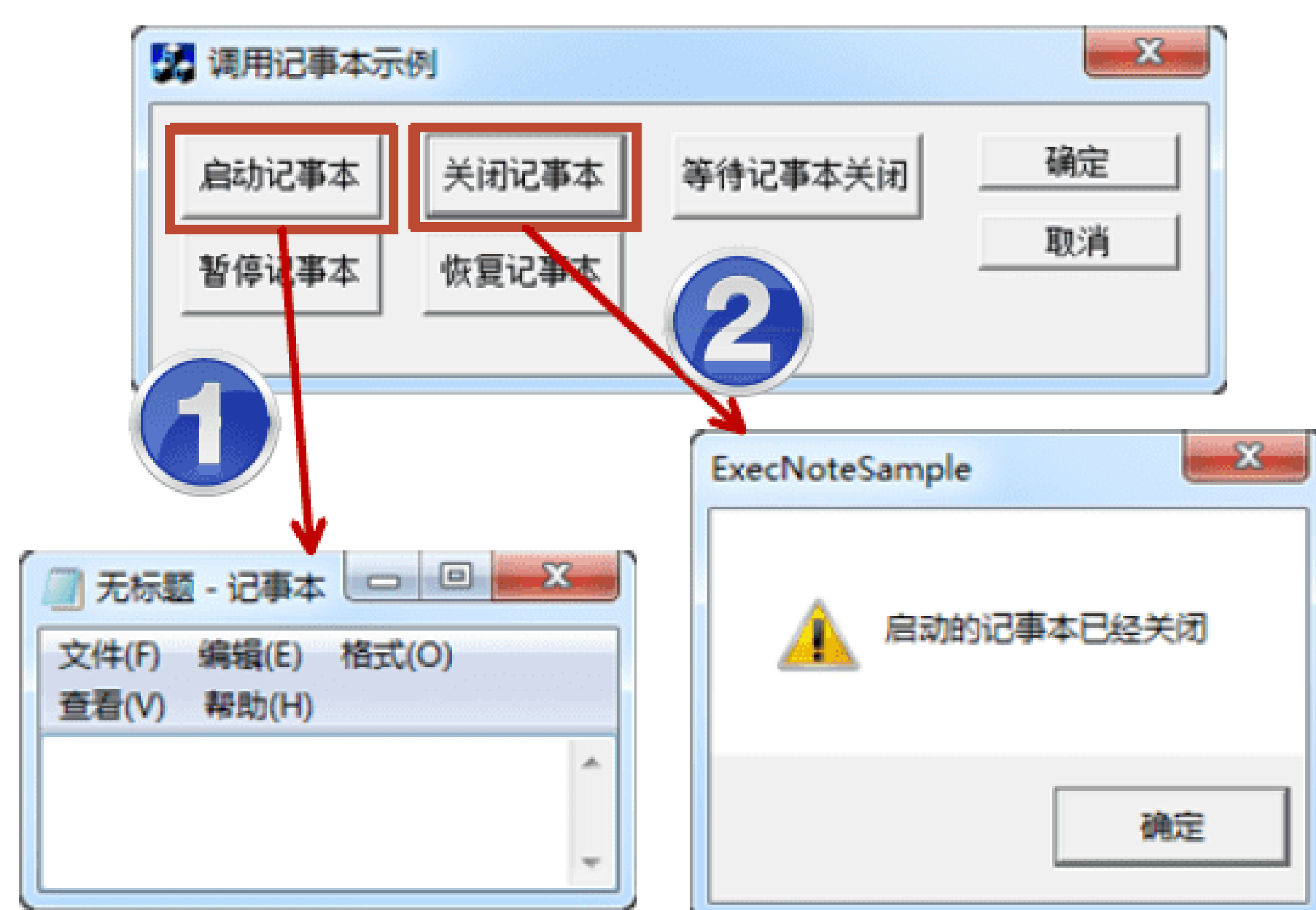


图 23-9　启动记事本并控制其关闭的程序运行效果

23.3.8　调用记事本程序并挂起

在 23.3.7 小节的基础上，可以增加暂停记事本程序和恢复记事本程序运行的功能。主要是在记事本线程的运行代码中，通过判断 bRunning 变量的值实现。如果要暂停记事本程序，则调用 SuspendThread()函数挂起记事本程序对应的主线程；如果要恢复记事本程序，则调用 ResumeThread()函数恢复记事本程序对应的主线程。程序主进程中的处理代码如下：

```
01  void CexecNoteSampleDlg::OnButtonStop()
02  {
03      bRunning = 2;
04  }
05  void CexecNoteSampleDlg::OnButtonRestore()
06  {
07      bRunning = 3;
08  }
```

在上面代码中，OnButtonStop()函数是暂停记事本程序的代码，将 bRunning 的状态值设置为 2，则在记事本线程中的主循环中会判断 bRunning 的值是否为 2。如果是，会挂起记事本线程。同样 OnButtonRestore()函数是恢复记事本程序的代码，将 bRunning 的状态值设置为 3。当单击“暂停记事本”按钮时，打开的记事本程序会暂时不响应；当单击“恢复记事本”按钮时，打开的记事本程序会恢复工作。

23.3.9　监测记事本程序关闭

在某些程序中，需要在打开记事本程序并等待记事本被用户关闭后，提示程序执行相应的操作。本小节介绍此种方式的实现。这时，线程工作函数在打开记事本程序后，需要调用 WaitForSingleObject()函数等待打开的记事本程序的进程 ID 无效，表示记事本程序被关闭，此时就可以释放相关资源，提示用户并退出线程。代码如下：

```
01  void CexecNoteSampleDlg::OnButtonWait()              //等待处理函数
02  {
03      pStartAndWaitThread = new CWinThread();          //创建等待线程
04      //设置是否自动删除为 false
05      pStartAndWaitThread->m_bAutoDelete = false;
06      //启动线程
07      pStartAndWaitThread=AfxBeginThread(StartAndWaitThreadProc,NULL);
08      //输出信息
09      if (pStartAndWaitThread == NULL)
10          MessageBox("启动记事本失败");
11  }
12  UINT StartAndWaitThreadProc( LPVOID pParam )
13  {                                            //启动记事本程序并等待其结束
14      STARTUPINFO si;                                  //启动信息结构
15      PROCESS_INFORMATION pi;                          //进程信息
16      ZeroMemory( &si, sizeof(si) );                   //初始化结构变量
17      si.cb = sizeof(si);                              //赋值结构长度
18      //启动进程
19      if( !CreateProcess(NULL,"notepad.exe",NULL,NULL,false,0,
20          NULL,NULL,&si,&pi))
```

```
21          AfxMessageBox("启动记事本进程失败.");
22      WaitForSingleObject( pi.hProcess, INFINITE );//无限期等待进程结束
23      CloseHandle( pi.hProcess );             //关闭进程
24      CloseHandle( pi.hThread );              //关闭线程
25      AfxMessageBox("启动的记事本已经关闭");
26      return 0;
27  }
```

上面代码创建完记事本进程后，调用 WaitForSingleObject()函数等待进程关闭。当用户关闭打开的记事本程序后，运行效果如图 23-10 所示。

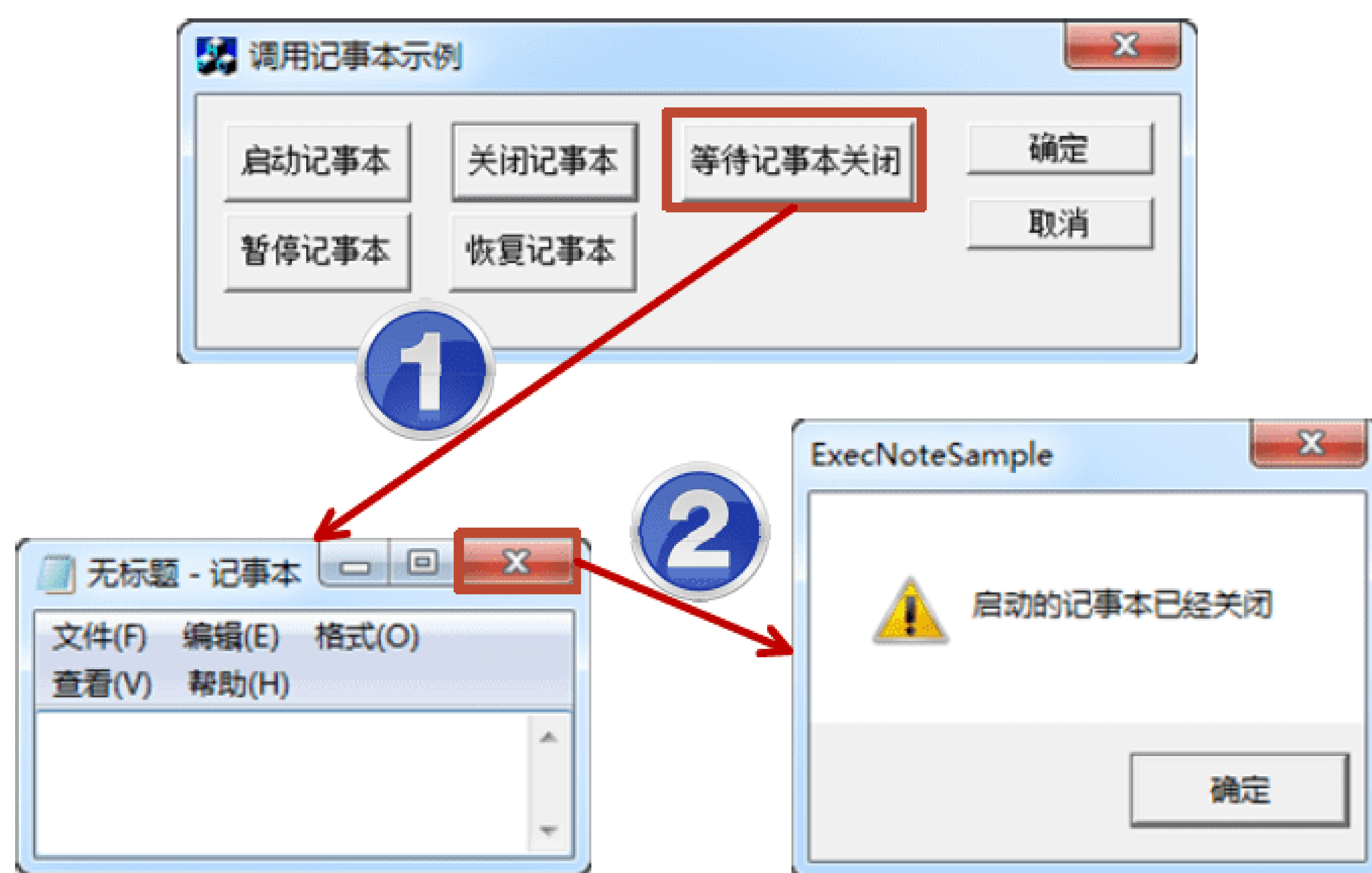


图 23-10　等待打开的记事本关闭的程序运行效果

23.4　线程间的通信

在程序中，经常会遇到线程间的数据通信。实现线程间通信的方式有多种，其中常用的主要有两种，一种是通过全局变量实现线程间的通信，一种是使用自定义消息实现线程间的通信。本节将介绍这两种线程间通信的实现方法。

23.4.1　使用全局变量

使用全局变量实现线程间通信是最简单的实现线程间通信的方式。核心做法是在程序中定义全局变量，各个线程都可以直接访问此全局变量的值，从而达到线程间通信的目的。在实际编程中，此种方式主要用在以下两种情况下。

- ❑ 使用全局变量控制线程的工作状态，如在 23.3.7 小节中的打开记事本示例，通过整型全局变量 bRunning 判断线程的工作状态。如果 bRunning 的值为 0，则线程会结束运行；如果 bRunning 的值为 1，则线程会正常运行；如果 bRunning 的值为 2，则线程会挂起；如果 bRunning 的值为 3，则线程会恢复运行。
- ❑ 通过全局变量实现各个线程操作同一对象。如在 23.1 节引入的银行中间件示例中，可以定义接收数据缓冲区，可以由接收线程在接收到数据后，将接收的数据存入此缓冲区，同时也可以由数据解析线程从中读取数据进行解析。

在使用全局变量实现线程间通信时，需要注意的是，当线程读取全局变量时，要保证此全局变量当前是有效的。现在重新看 23.3.1 小节中的例子，其中加入了 Sleep()函数。将其修改后，代码如下：

```
01  DWORD WINAPI ThreadFunc( LPVOID lpParam )        //线程函数
02  {
03      printf("启动线程.参数值=%d\n", lpParam);       //输出提示信息
04      return 0;
05  }
06  int main(int argc, char* argv[])                 //程序入口函数
07  {
08      DWORD dwThreadId;                            //线程 ID 变量
09      DWORD dwValue = 7;                           //参数变量
10      HANDLE hThread;                              //线程句柄
11      hThread = CreateThread(NULL,0,
12          (LPTHREAD_START_ROUTINE)ThreadFunc,
13          (LPVOID) dwValue, 0, &dwThreadId);       //创建线程
14      if (hThread == NULL)
15          printf("创建线程失败");                     //输出信息
16      Sleep(2000);                                 //延时
17      CloseHandle( hThread );                      //关闭句柄
18      printf("主线程结束\n");                       //输出提示信息
19      return 0;
20  }
```

参看上面的代码，如果省掉 Sleep 语句，则程序的运行结果会与预期的不一致——程序只是在屏幕上打印“主线程结束”。这是因为当将参数值 dwValue 传入线程函数后，主线程并不会停止，而是继续运行。如果没有 Sleep 语句，则可能在主线程运行完成时，子线程还没有执行，因此结果是不可预知的。所以，使用全局变量时要注意其处理。程序运行效果如图 23-11 所示。

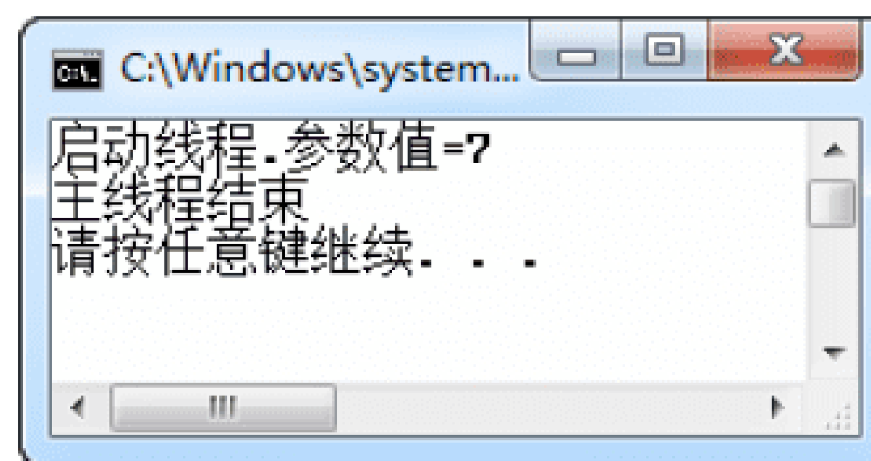


图 23-11　使用全局变量实现线程间的通信运行效果

23.4.2　使用自定义的消息

除了使用全局变量外，还可以使用自定义消息实现线程间的通信。通常情况下，此种方式用在线程将数据传递给进程的情况。下面是一个示例，在主线程中创建一个实现每隔 1 秒自增 1 的线程。每当当前计数值是 2 的整数倍时，从线程中发送消息给主进程，并将当前的计数值通过消息参数发送给主进程，然后通过主进程在界面上显示出来。代码如下：

```
01  void CthreadCommSampleDlg::OnButtonStart()  //启动线程函数
02  {
03      pThread = new CWinThread();              //创建线程对象
04      pThread->m_bAutoDelete = false;          //设置是否自动删除为 false
05      pThread = AfxBeginThread(MyThreadProc, this->m_hWnd);//启动线程
06      if (pThread != NULL)
07          WriteLog("启动线程成功");              //输出错误信息
08  }
```

上面代码创建运行 MyThreadProc()函数的线程，并在日志编辑框中显示创建线程成功的提示。下面的代码是线程的运行函数。

```
01  bool bThreadRunning = true;                  //初始化线程运行状态为 true
02  UINT MyThreadProc( LPVOID pParam )           //线程处理函数
03  {
04      int globalCounter = 0;                   //定义全局计数变量
05      while (bThreadRunning)                   //循环增加计数变量
06      {
07          globalCounter++;                     //计数变量自增 1
08          if (globalCounter > 99999999)
09              globalCounter=0;                 //计数大于最大值，归 0
10          if ((globalCounter % 2) == 0)        //是 2 的倍数，发送界面消息
11              ::PostMessage((HWND)pParam, WM_THREAD_TO_PROCESS,
12              globalCounter, 0);
13          Sleep(1000);                         //延时处理
14      }
15      return 0;
16  }
```

在上面代码中，MyThreadProc()函数用于定义线程的处理代码，根据 bThreadRunning 判断线程是否继续执行。如果此值为 true，则线程继续执行；如果此值为 false，则线程结束。在 while 循环中，每隔 1 秒执行一次自增 1，并判断当前值是否为 2 的整数倍。如果是 2 的整数倍，则发送消息给进程，并将当前计数值通过消息参数发送给主进程。

```
01  LRESULT CThreadCommSampleDlg::OnThreadMsg(WPARAM wParam,
02        LPARAM lParam)
03  {
04      CString log;
05      log.Format("\r\n 从线程传递过来的数据，当前为=%u", wParam);
06      WriteLog(log);
07      return 1;
08  }
```

上面代码是主进程对接收到的线程的消息的处理函数，将消息传递过来的第一个参数，即线程中的计数值显示在日志编辑框中。使用自定义消息实现线程之间的通信，可以完成工作线程与界面之间的通信。如当工作线程完成到一定进度后，可以通过自定义消息的方式将当前进度发送给对话框，再由对话框将当前进度以图形化的方式显示给用户，以完成处理和界面的统一，这也是 Windows 操作系统的特色之一。程序运行效果如图 23-12 所示。

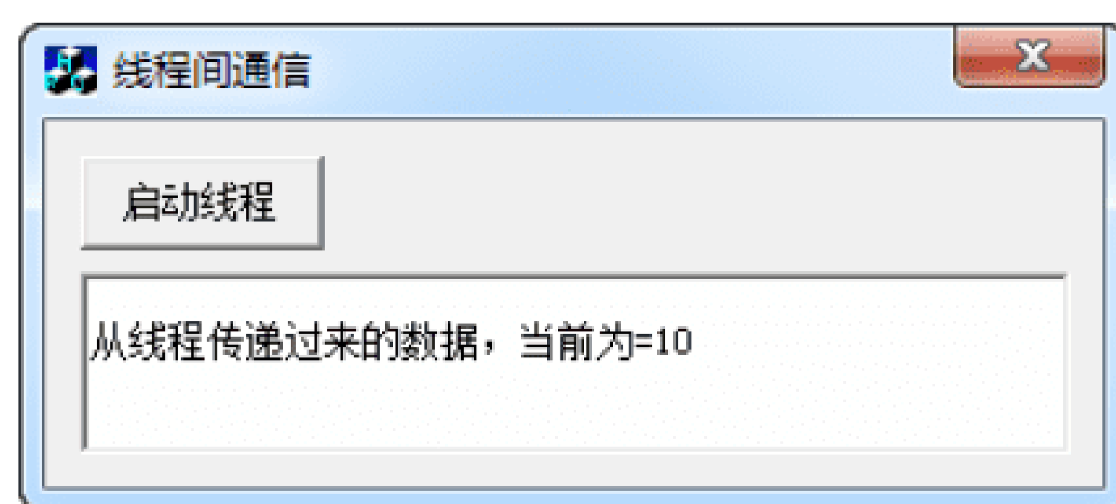


图 23-12　使用自定义的消息实现线程间的通信

23.5　线程的同步

在 23.4 节中介绍了有关线程间通信的实现方法。从中可以看出，在线程间进行通信，经常需要传递全局变量或指定内存块，由其他线程或进程进行操作，此时就存在一个线程同步的问题。当多个线程或进程同时访问同一块存储空间时，如何保证不会发生冲突，就

成为多线程编程的关键。本节就介绍如何在 VC 中实现线程同步。

23.5.1　等待函数

对于单线程程序，不存在对象访问同步问题。因为当前只有一个线程访问对象，不会发生多个线程同时访问同一个对象的情况。但是，在多线程程序中，则存在各个线程之间建立“良好合作关系”，彼此协商处理对象的问题。对于对象的读取，可以不使用同步处理，但是只要涉及到对象的写操作，必须实现线程同步的处理。Win32 API 为多线程技术提供了一组等待函数阻塞线程执行，直到符合一定的条件，线程才会继续执行。其中包括 3 种类型的等待函数。

（1）单对象等待函数，是指与单个对象的同步相关的操作函数，包括 SignalObjectAndWait()函数、WaitForSingleObject()函数和 WaitForSingleObjectEx()函数，这些函数需要一个同步对象的句柄。这些函数当遇到下面几种情况时会返回。

- 当指定的对象进入终止状态时。
- 当超过等待超时时间时。设置超时时间为 INFINITE 可以使得等待过程不考虑超时时间，一直等待下去。

SignalObjectAndWait()函数使得调用线程自动地设置对象的状态为有信号，并等待其他对象的状态被设置为有信号的。

（2）多对象等待函数，主要处理多个线程与多个对象之间的同步问题。包括 WaitForMultipleObjects()函数、WaitForMultipleObjectsEx()函数、MsgWaitForMultipleObjects()函数和 MsgWaitForMultipleObjectsEx()函数。这些函数使得调用线程指定一个包含一个或多个同步对象句柄的数组。这些函数当遇到下面几种情况时会返回。

- 当这些指定对象中的任何一个被设置为有信号的或所有对象的状态被设置为有信号时。读者可以在函数调用中控制是等待一个对象还是等待所有对象的状态信号。
- 过了等待超时时间。设置等待超时时间为 INFINITE 可以使得等待过程不考虑超时时间，一直等待下去。

MsgWaitForMultipleObjects()函数和 MsgWaitForMultipleObjectsEx()函数允许在对象句柄数组中指定输入事件对象。在线程的输入队列中，指定等待的输入类型。如使用 MsgWaitForMultipleObjects()函数阻塞执行，直到指定对象被设置为有信号，并且在线程的输入队列中共用可用的鼠标输入。线程可以使用 GetMessage()函数或 PeekMessage()函数获取输入。当所有等待对象均设置为有信号时，系统会将信号状态通知等待对象。

（3）可报警的等待函数，可以完成带有报警功能的等待功能。它是对上面两种等待函数的扩展应用，主要包括 MsgWaitForMultipleObjectsEx()函数、SignalObjectAndWait()函数、WaitForMultipleObjectsEx()函数和 WaitForSingleObjectEx()函数。这些函数可以选择完成可警告的等待操作。在可警告的等待操作中，当遇到指定条件时，函数会返回，但是如果 I/O 完成程序的系统队列或 APC 可执行程序被退出等待线程，也会返回。

等待函数直到遇到指定的关键部分，才会返回。等待函数的类型确定了使用的关键段。当调用等待函数时，会检查是否遇到了等待的关键部分。如果还没有遇到关键段，调用线程进入有效的等待状态，在等待遇到关键部分时，只消耗很小的处理时间。

在返回前，等待函数可以修改一些同步对象的状态。只能修改那些设置为有信号状态

后可以导致函数返回的对象。

- 每次信号量对象的次数减少一次，如果计数为 0，则信号量的状态设置为无信号的。
- 互斥对象的状态、自动重置事件和修改通知对象被设置为无信号。
- 同步计时器的状态被设置为无信号。
- 手动重置事件、手动重置定时器、进程、线程和控制台输入对象不受等待函数的影响。

下面以 WaitForMultipleObjects()函数为例，介绍等待函数的常用参数，此函数的函数原型为：

```
DWORD WaitForMultipleObjects(
  DWORD nCount,                    //要等待的对象的个数
  CONST HANDLE *lpHandles,         //表示要等待的对象的数组
  BOOL fWaitAll,                   //表示是否等待所有的对象都是有信号状态
  DWORD dwMilliseconds );          //表示等待的超时时间
```

其中，fWaitAll 参数表示是否等待所有的对象都是有信号状态。如果此参数为 true，则表示当要等待 lpHandles 中的所有对象都是有信号状态时，才会返回；如果此参数为 false，则表示只要有一个对象处于有信号状态，函数就会返回。从 23.5.2 小节开始，将分别介绍多线程的同步对象及其使用。

23.5.2　利用事件对象

事件对象可以用于表示一个对象是否有信号。如果有信号，表示当前对象不被其他线程操作；如果其他线程对其进行操作，则对应的事件对象处于无信号状态。其他调用等待函数的线程会等待此事件对象处于有信号状态，才会继续执行。使用 CreateEvent()函数可以创建命名的或匿名的事件对象。其函数原型如下：

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
                                   //指向安全 SECURITY_ATTRIBUTES 结构的指针
  BOOL bManualReset,               //指定事件是否是手动事件对象
  BOOL bInitialState,              //指定事件对象的初始化状态
  LPCTSTR lpName );                //指定事件对象的名称
```

其中，bManualReset 参数用于指定事件是否是手动事件对象。如果此参数为 true，则用户必须使用 ResetEvent()函数手动将事件对象置为无信号状态；如果此参数为 false，当线程释放相应资源时，系统会自动重置事件对象的状态为无信号。

如果创建事件对象成功，则返回事件对象的句柄。如果要创建的命名事件对象已经存在，则可以通过 GetLastError()函数判断其值是否为 ERROR_ALREADY_EXISTS。如果创建事件对象失败，则返回值为 NULL。

创建完事件对象后，读者可以通过 SetEvent()函数设置事件对象的状态为有信号状态，通过 ResetEvent()函数设置事件对象的状态为无信号状态。这两个函数的参数都是 CreateEvent()函数返回的事件句柄。通过这两个函数，用户可以在事件对象的信号状态之间切换。需要注意的是，对于自动事件对象，当线程从等待函数中返回并获得资源时，系统会自动调用 ResetEvent()函数，将事件对象的状态置为无信号状态，而不需要手动调用 ResetEvent()函数。

23.5.3　使用事件对象实例

23.5.2 小节介绍了如何利用事件对象实现线程同步，本小节以一个实例介绍事件对象的使用。在本示例中，创建两个线程，均用于判断全局变量 iGolbalCount 的值是否小于指定值。如果小于，则在屏幕上显示并将 iGolbalCount 值自增 1，直到 iGolbalCount 的值大于指定值，线程才会结束运行。代码如下：

```
01  DWORD WINAPI ThreadProc1(LPVOID lpParam);              //线程处理函数 1
02  DWORD WINAPI ThreadProc2(LPVOID lpParam);              //线程处理函数 2
03  int iGolbalCount=0;                                    //全局计数变量
04  int iMax = 12;                                         //最大值变量
05  HANDLE hEvent;                                         //事件句柄
06  int main(int argc, char* argv[])                       //主函数
07  {
08      HANDLE hThread1, hThread2;                         //线程句柄
09      //创建事件
10      hEvent=CreateEvent(NULL,false,false,LPCTSTR("iGolbalCount"));
11      if (hEvent == NULL)                                //判断创建结果
12      {
13          printf("创建事件对象失败！\r\n");
14          return 0;
15      }
16      SetEvent(hEvent);
17      hThread1=CreateThread(NULL,0,ThreadProc1,NULL,0,NULL);
18      hThread2=CreateThread(NULL,0,ThreadProc2,NULL,0,NULL);
19      Sleep(50000);                                      //延时
20      CloseHandle(hEvent);                               //关闭事件对象句柄
21      printf("主线程结束!\n");                           //输出提示信息
22      return 0;
23  }
```

上面代码声明了两个线程的执行函数——ThreadProc1()和 ThreadProc2()。定义了全局计数变量 iGolbalCount 和计数最大值变量 iMax，定义了事件对象 hEvent。主函数所做的工作包括创建事件对象、设置事件对象为有信号状态、创建两个线程、主线程休眠一定时间、关闭事件句柄以及提示线程结束并返回。

```
01  DWORD WINAPI ThreadProc1(LPVOID lpParam)          //线程 1 处理函数
02  {
03      while (true)
04      {
05          WaitForSingleObject(hEvent,INFINITE);     //等待事件对象
06          if (iGolbalCount < iMax)                  //增加计数
07          {
08              printf("这里是线程 1，iGolbalCount=%d\r\n",iGolbalCount++);
09              SetEvent(hEvent);
10          }
11          else
12          {
13              SetEvent(hEvent);
14              break;
15          }
16          Sleep(10);
```

```
17      }
18      return 0;
19  }
20  DWORD WINAPI ThreadProc2(LPVOID lpParameter)   //线程 2 处理函数
21  {
22      while (true)
23      {
24          WaitForSingleObject(hEvent,INFINITE);   //等待事件对象
25          if (iGolbalCount < iMax)                //增加计数
26          {
27              printf("这里是线程 2，iGolbalCount=%d\r\n",iGolbalCount++);
28              SetEvent(hEvent);
29          }
30          else
31          {
32              SetEvent(hEvent);
33              break;
34          }
35          Sleep(10);
36      }
37      return 0;
38  }
```

上面代码定义了线程函数的实现。主体是一个 while 循环，在循环中，首先调用 WaitForSingleObject()函数阻塞线程，当 hEvent 事件对象处于有信号状态时，线程才继续执行。当等待函数返回时，由于 hEvent 创建时指定的是自动事件，因此，系统会自动设置事件对象为无信号状态。此时，其他线程在 WaitForSingleObject()函数处会等待，直到此线程运行到 SetEvent()调用后，其他线程才会继续执行。线程从等待函数返回后，会根据当前计数值的取值执行相应的操作。但是不管在何种情况下，在执行完后，都会调用 SetEvent()函数，设置事件对象的状态为有信号状态。程序运行的效果如图 23-13 所示。

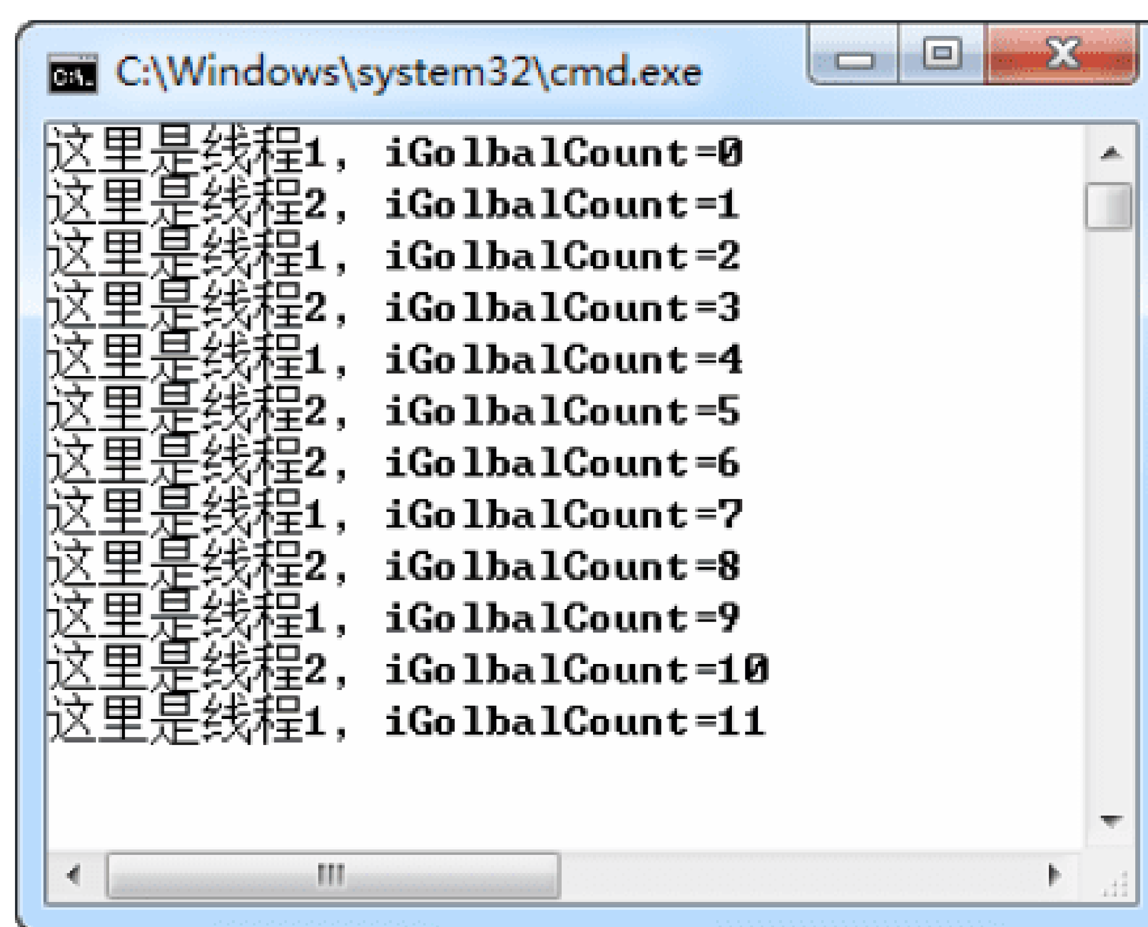


图 23-13　利用事件对象实现线程同步的运行效果

从图 23-13 中可以看出，操作系统对线程的调度在同一优先级的情况下是随机的，因此，此程序可能每次运行的结果都不同。虽然计数值的每次结果是一样的，但是在哪个线程中执行的自增 1 是不确定的。

23.5.4　利用临界区

临界区是允许一个线程在同一时间访问一个资源或关键源代码的同步对象。如增加结点到链表中，在同一时间只允许一个线程处理。通过使用 CRITICAL_SECTION 对象控制链表，同一时间只有一个线程获取访问链表的权限。临界区又称为关键段，意思为保护关键资源的访问。使用临界区前，需要先调用 InitializeCriticalSection()函数初始化临界区对象。其函数原型为：

```
VOID InitializeCriticalSection(
LPCRITICAL_SECTION lpCriticalSection); //表示要初始化的临界区的指针
```

使用完临界区后，需要调用 DeleteCriticalSection()函数释放临界区中的资源，其函数原型为：

```
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

此函数释放空闲关键段中的所有资源。其中，lpCriticalSection 参数表示要释放资源的临界区的指针。使用临界区进行线程同步，通过 EnterCriticalSection()函数和 LeaveCriticalSection()函数进入关键段和离开关键段，也可以理解为锁定关键段和解锁关键段。这两个函数的原型为：

```
VOID EnterCriticalSection( LPCRITICAL SECTION lpCriticalSection);
VOID LeaveCriticalSection( LPCRITICAL SECTION lpCriticalSection);
BOOL TryEnterCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

其中，lpCriticalSection 参数用作关键段的标识。而 TryEnterCriticalSection()函数会试图进入关键段，如果关键段当前不可进入，则不会阻塞等待，而直接返回 false。如果可以进入关键段，则返回 true。

23.5.5　利用临界区实例

23.5.4 小节介绍了如何利用临界区实现线程同步，本小节以一个实例介绍临界区的使用。本示例实现的功能与 23.5.3 小节中的示例是相同的。代码如下：

```
01  DWORD WINAPI ThreadProc1(LPVOID lpParam);          //线程处理函数 1
02  DWORD WINAPI ThreadProc2(LPVOID lpParam);          //线程处理函数 2
03  int iGolbalCount=0;                                //全局计数变量
04  int iMax = 12;                                     //最大值变量
05  CRITICAL_SECTION cs;                               //临界区句柄
06  int main(int argc, char* argv[])                   //主函数
07  {
08      HANDLE hThread1, hThread2;                     //线程句柄
09      InitializeCriticalSection(&cs);                //初始化临界区对象
10      hThread1=CreateThread(NULL,0,ThreadProc1,NULL,0,NULL);
11      hThread2=CreateThread(NULL,0,ThreadProc2,NULL,0,NULL);
12      Sleep(50000);                                  //延时
13      DeleteCriticalSection(&cs);                    //删除临界区对象
14      printf("主线程结束!\n");                        //输出提示信息
15      return 0;
16  }
```

上面代码声明了两个线程的执行函数 ThreadProc1()和 ThreadProc2()。定义了全局计数变量 iGolbalCount 和计数最大值变量 iMax，定义了临界区对象 cs。主函数所做的工作包括初始化临界区对象、创建两个线程、主线程休眠一定时间、删除临界区对象以及提示线程结束并返回。

```
01  DWORD WINAPI ThreadProc1(LPVOID lpParam)       //线程 1 处理函数
02  {
03      while (true)                               //执行 while 循环
04      {
05          EnterCriticalSection(&cs);             //进入临界区
06          if (iGolbalCount < iMax)               //增加计数
```

```
07          {
08              printf("这里是线程 1，iGolbalCount=%d\r\n", iGolbalCount++);
09              LeaveCriticalSection(&cs);              //离开临界区
10          }
11          else
12          {
13              LeaveCriticalSection(&cs);              //离开临界区
14              break;
15          }
16          Sleep(10);                                  //延时
17      }
18      return 0;
19  }
20  DWORD WINAPI ThreadProc2(LPVOID lpParameter)    //线程 2 处理函数
21  {
22      while (true)                                    //执行 while 循环
23      {
24          EnterCriticalSection(&cs);                  //进入临界区
25          if (iGolbalCount < iMax)                    //增加计数
26          {
27              printf("这里是线程 2，iGolbalCount=%d\r\n", iGolbalCount++);
28              LeaveCriticalSection(&cs);              //离开临界区
29          }
30          else
31          {
32              LeaveCriticalSection(&cs);              //离开临界区
33              break;
34          }
35          Sleep(10);                                  //延时
36      }
37      return 0;
38  }
```

上面代码定义了线程函数的实现。主体是一个 while 循环，在循环中，首先调用 EnterCriticalSection()函数等待获取关键资源的访问权。当等待函数返回时，会根据当前计数值的取值执行相应的操作。但是不管在何种情况下，在执行完后，都会调用 LeaveCriticalSection()函数退出临界区，以释放关键资源，使得其他线程可以访问关键资源。程序运行的效果如图 23-14 所示。

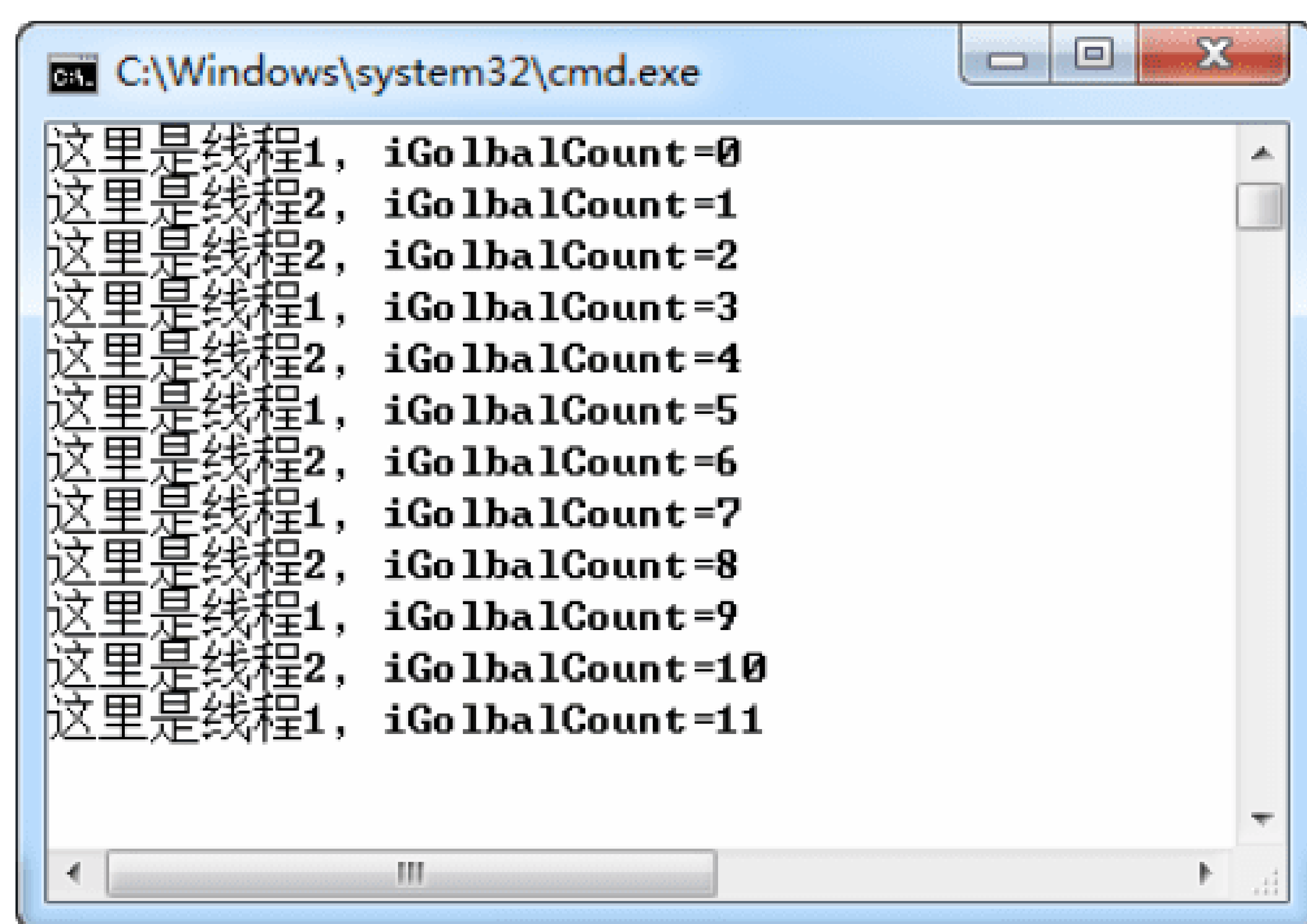


图 23-14　利用临界区实现线程同步的运行效果

23.5.6　利用信号量

信号量是一个同步对象，允许一定数目的线程同时访问一个或多个进程的资源。信号量对象用于指定可以同时访问资源的线程的最大个数和初始化时可以访问资源的线程数。此对象对于只能由有限数目的用户访问的共享资源的访问控制很有用。需要注意的是，当对共享资源进行写操作时，信号量的个数不能超过 1。使用 CreateSemaphore()函数可以创建信号量对象，其函数原型为：

```
HANDLE CreateSemaphore(
  LPSECURITY ATTRIBUTES lpSemaphoreAttributes,
                                //指向 SECURITY ATTRIBUTES 结构的指针
  LONG lInitialCount,           //指定信号量对象初始状态下的信号量个数
  LONG lMaximumCount,           //指定信号量对象最多允许多少个线程同时访问共享资源
  LPCTSTR lpName);              //指定信号量对象的名称
```

如果创建事件对象成功，则返回事件对象的句柄。如果要创建的命名事件对象已经存在，则可以通过 GetLastError()函数判断其值是否为 ERROR_ALREADY_EXISTS。如果创建事件对象失败，则返回值为 NULL。

使用 OpenSemaphore()函数可以打开已经创建的信号量对象，其函数原型为：

```
HANDLE OpenSemaphore(
  DWORD dwDesiredAccess,
                         //指定信号量对象的访问权限，可以指定是否可以修改信号量的值
  BOOL bInheritHandle,   //指定信号量对象是否可以被继承
  LPCTSTR lpName );      //指定要打开的信号量对象的名称
```

当完成对共享资源的访问后，需要调用 ReleaseSemaphore()函数增加信号量对象中的信号量个数，使得访问共享资源的线程增加。其函数原型为：

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,     //指定操作的信号量对象句柄
  LONG lReleaseCount,    //指定要向信号量对象中增加的信号量个数
  LPLONG lpPreviousCount);
                         //存放在执行此操作前信号量对象中的信号量个数
```

如果操作成功，函数返回 true；否则，返回 false。

23.5.7　利用信号量实例

23.5.6 小节介绍了如何利用信号量实现线程同步，本小节以一个示例介绍信号量的使用。本示例实现的功能与 23.5.3 小节中的示例是相同的。代码如下：

```
01  DWORD WINAPI ThreadProc1(LPVOID lpParam);           //线程处理函数 1
02  DWORD WINAPI ThreadProc2(LPVOID lpParam);           //线程处理函数 2
03  int iGolbalCount=0;                                 //全局计数变量
04  int iMax = 12;                                      //最大值变量
05  int cMax = 1;                                       //字符最大值
06  HANDLE hSemaphore;                                  //信号量句柄
07  int main(int argc, char* argv[])
08  {
09      HANDLE hThread1, hThread2;
10      hSemaphore = CreateSemaphore( NULL, cMax, cMax, NULL);
11      if (hSemaphore == NULL)
12      {
13          printf("创建信号量对象失败");
14          return 0;
15      }
16      hThread1=CreateThread(NULL,0,ThreadProc1,NULL,0,NULL);
17      hThread2=CreateThread(NULL,0,ThreadProc2,NULL,0,NULL);
18      Sleep(5000);                                    //延时
19      printf("主线程结束!");                            //输出提示信息
20      return 0;
21  }
```

上面代码声明了线程的执行函数 ThreadProc1()和 ThreadProc2()。定义了全局计数变量 iGolbalCount 和计数最大值变量 iMax，定义了信号量对象 hSemaphore 和信号量中最大的信号量数目，即同时允许访问资源的线程个数。主函数所做的工作包括创建信号量对象、创建两个线程、主线程休眠一定时间和提示线程结束并返回。

```
01  DWORD WINAPI ThreadProc1(LPVOID lpParam)          //线程 1 处理函数
02  {
03      while (true)                                  //执行 while 循环
04      {
05          DWORD dwWaitResult = WaitForSingleObject(hSemaphore, 0L);
06          //等待对象信号量
07          if (dwWaitResult == WAIT_OBJECT_0)
08          {
09              if (iGolbalCount < iMax)              //增加全局变量值
10              {
11                  printf("这里是线程 1，iGolbalCount=%d\r\n",
12                          iGolbalCount++);
13                  ReleaseSemaphore(hSemaphore, 1, NULL);//释放信号量
14              }
15              else
16              {
17                  ReleaseSemaphore(hSemaphore, 1, NULL);//释放信号量
18                  break;
19              }
20          }
21          Sleep(10);
22      }
23      return 0;
24  }
25  DWORD WINAPI ThreadProc2(LPVOID lpParameter)      //线程 2 处理函数
26  {
27      while (true)                                  //执行 while 循环
28      {
29          DWORD dwWaitResult = WaitForSingleObject(hSemaphore, 0L);
30          //等待对象信号量
31          if (dwWaitResult == WAIT_OBJECT_0)        //增加全局变量值
32          {
33              if (iGolbalCount < iMax)
34              {
35                  printf("这里是线程 2，iGolbalCount=%d\r\n",
36                           iGolbalCount++);
37                  ReleaseSemaphore(hSemaphore, 1, NULL);//释放信号量
38              }
39              else
40              {
41                  ReleaseSemaphore(hSemaphore, 1, NULL);//释放信号量
42                  break;
43              }
44          }
45          Sleep(10);
46      }
47      return 0;
48  }
```

上面代码中定义了线程函数的实现。主体是一个 while 循环，在循环中，首先调用

WaitForSingleObject()函数等待信号量有信号。当等待函数返回后，判断是否获得信号量。如果获得信号量，则会根据当前计数值的取值执行相应的操作。但是不管在何种情况下，在执行完后，都会调用 ReleaseSemaphore()函数释放占用的信号量。此时信号量句柄的信号量个数增加 1，使得当前可访问信号量句柄的线程又增加了一个。从中可以看出，信号量是控制同时访问资源的线程的个数不要超过指定个数，因此，此处需要修改变量的值。所以，只能设置同时访问个数为 1，否则，系统会发生资源访问冲突。程序运行的效果如图 23-15 所示。

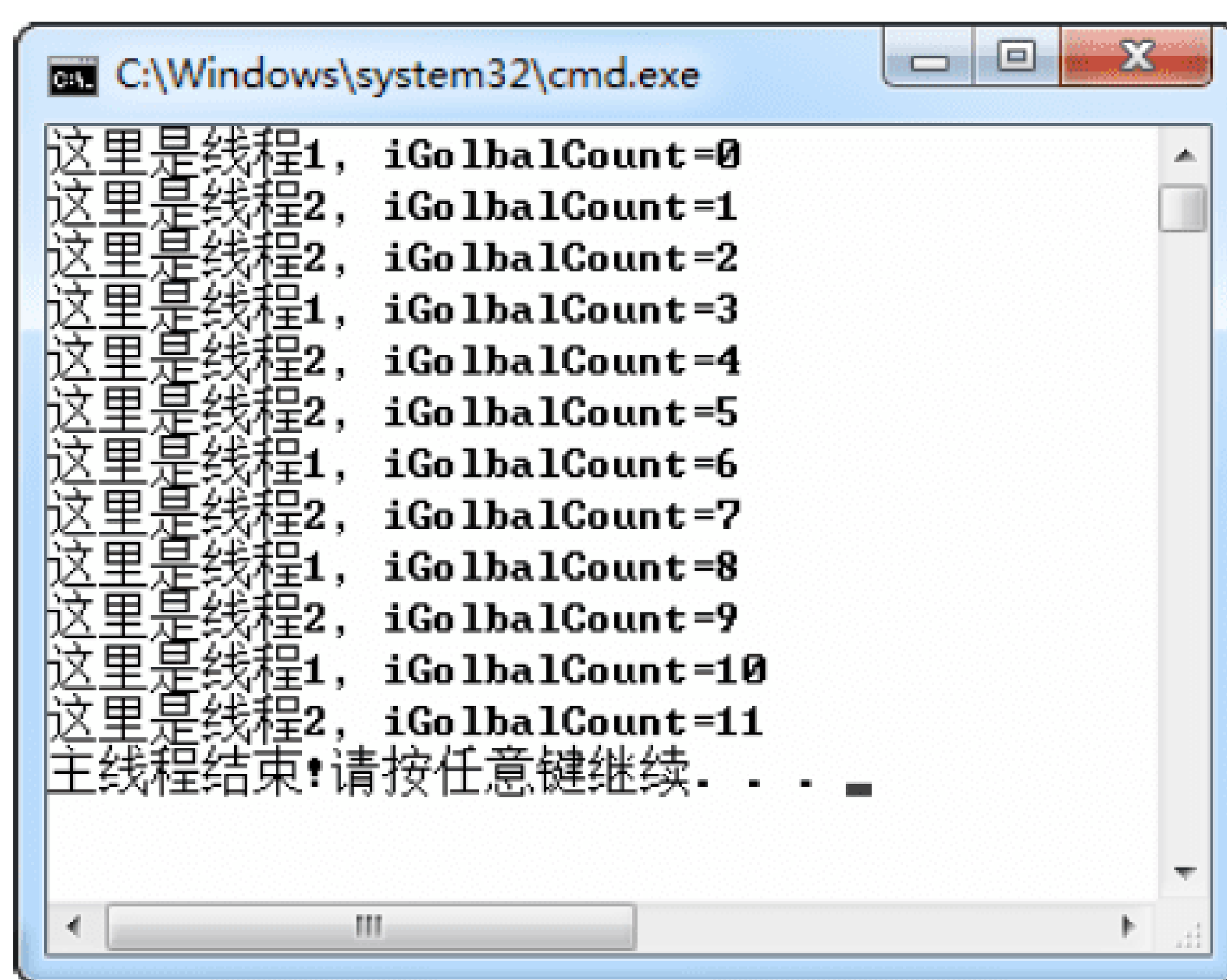


图 23-15　利用信号量实现线程同步的运行效果

23.5.8　利用互斥对象

互斥对象可以用于表示线程对互斥对象的占有权，其他调用等待函数的线程会等待所有使用此互斥对象的线程释放占有权后，才会继续执行。使用 CreateMutex()函数可以创建命名的或匿名的互斥对象。其函数原型如下：

```
HANDLE CreateMutex (
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
                                  //指向 SECURITY_ATTRIBUTES 结构的指针
  BOOL bInitialOwner,             //指定当前线程是否具有互斥对象的占有权
  LPCTSTR lpName );               //指定互斥对象的名称
```

其中，bInitialOwner 参数用于指定当前线程是否具有互斥对象的占有权。如果此参数为 true，则当前线程具有对此互斥对象的占有权；如果此参数为 false，则当前线程不具有对此互斥对象的占有权。如果创建互斥对象成功，则返回互斥对象的句柄。如果要创建的互斥对象已经存在，则可以通过 GetLastError()函数判断其值是否为 ERROR_ALREADY_EXISTS。如果创建互斥对象失败，则返回值为 NULL。

创建完互斥对象后，用户可以通过调用等待函数等待互斥对象是自由可用的，从等待函数返回后，线程就具有对互斥对象的占有权。在互斥部分操作完成后，可以调用 ReleaseMutex()函数释放对互斥对象的占用，使得其他等待此互斥对象的线程可以继续运行。

23.5.9　利用互斥对象实例

23.5.8 小节介绍了如何利用互斥对象实现线程同步，本小节以一个实例介绍互斥对象的使用。在本示例中，创建两个线程，均用于判断全局变量 iGolbalCount 的值是否小于指定值。如果小于，则在屏幕上显示并将 iGolbalCount 值自增 1，直到 iGolbalCount 的值大于指定值，线程才会结束运行。代码如下：

```
01  DWORD WINAPI ThreadProc1(LPVOID lpParam);          //线程处理函数 1
02  DWORD WINAPI ThreadProc2(LPVOID lpParam);          //线程处理函数 2
```

```
03  int iGolbalCount=0;                             //全局计数变量
04  int iMax = 12;                                  //最大值变量
05  HANDLE hMutex;                                  //互斥对象句柄
06  int main(int argc, char* argv[])                //主函数
07  {
08      HANDLE hThread1, hThread2;                  //线程句柄
09      hThread1=CreateThread(NULL,0,ThreadProc1,NULL,0,NULL);
10      hThread2=CreateThread(NULL,0,ThreadProc2,NULL,0,NULL);
11      hMutex=CreateMutex(NULL,true,LPCTSTR("iGolbalCount"));
12      if (hMutex == NULL)                         //判断创建结果
13      {
14          printf("创建互斥对象失败！\r\n");
15          return 0;
16      }
17      WaitForSingleObject(hMutex,INFINITE);       //等待互斥对象
18      ReleaseMutex(hMutex);                       //释放互斥对象
19      ReleaseMutex(hMutex);                       //释放互斥对象
20      Sleep(5000);                                //延时
21      printf("主线程结束!\n");                     //输出提示信息
22      return 0;
23  }
```

上面代码声明了两个线程的执行函数 ThreadProc1()和 ThreadProc2()。定义了全局计数变量 iGolbalCount 和计数最大值变量 iMax，定义了互斥对象 hMutex。主函数所做的工作包括创建事件对象、设置互斥对象为有信号状态、创建两个线程、主线程休眠一定时间、关闭事件句柄以及提示线程结束并返回。

```
01  DWORD WINAPI ThreadProc1(LPVOID lpParam)        //线程 1 处理函数
02  {
03      while (true)                                //执行 while 循环
04      {
05          WaitForSingleObject(hMutex,INFINITE);   //等待互斥对象
06          //全局计数变量自增 1
07          if (iGolbalCount < iMax)
08              printf("这里是线程 1，iGolbalCount=%d\r\n",
09              iGolbalCount++);
10          else
11              break;
12          ReleaseMutex(hMutex);                   //释放互斥对象
13          Sleep(10);                              //延时
14      }
15      return 0;
16  }
17  DWORD WINAPI ThreadProc2(LPVOID lpParameter)    //线程 2 处理函数
18  {
19      while (true)                                //执行 while 循环
20      {
21          WaitForSingleObject(hMutex,INFINITE);   //等待互斥对象
22          //全局计数变量自增 1
23          if (iGolbalCount < iMax)
24              printf("这里是线程 2，iGolbalCount=%d\r\n",
25              iGolbalCount++);
26          else
27              break;
28          ReleaseMutex(hMutex);                   //释放互斥对象
29          Sleep(10);                              //延时
```

```
30      }
31      return 0;
32  }
```

上面代码定义了线程函数的实现。主体是一个 while 循环。在循环中，首先调用 WaitForSingleObject()函数阻塞线程，等待所有其他线程释放对互斥对象 hMutex 的访问，线程才继续执行。线程从等待函数返回后，会根据当前计数值的取值执行相应的操作。在执行完后，会调用 ReleaseMutex()函数，释放对互斥对象的所有权。需要注意的是，在退出 while 循环的时候没有调用 ReleaseMutex()函数，因此，在主函数中，后面要调用两次 ReleaseMutex()函数释放对互斥对象的访问。程序运行的效果如图 23-16 所示。

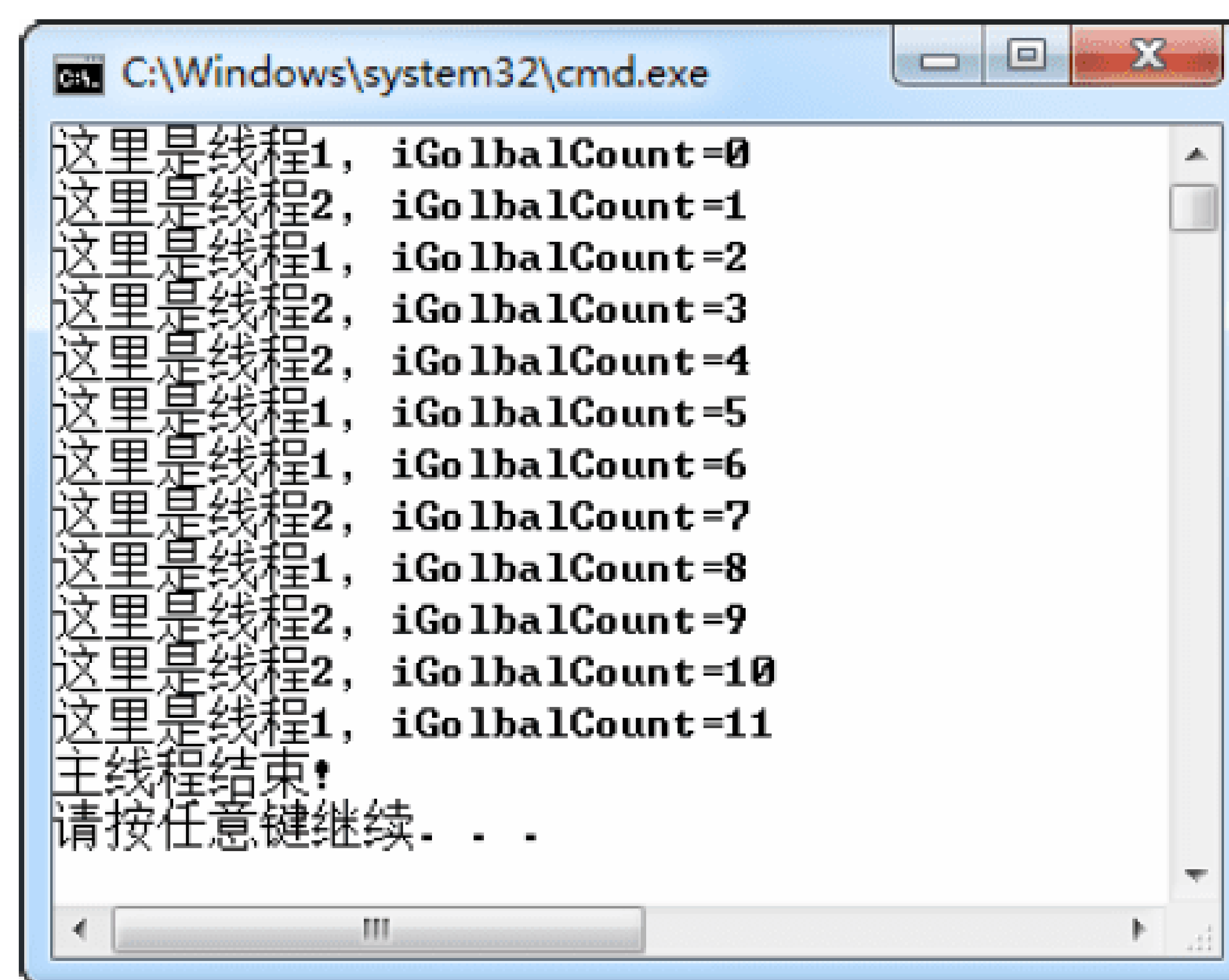


图 23-16　利用互斥对象实现线程同步的运行效果

23.6　多线程程序实例

本节结合一个实例，总结前面几节中介绍的有关多线程程序的开发。此实例创建两个线程，一个线程负责向全局变量的字符串列表中增加元素，元素的值是计数值与当前时间的组合；另一个线程负责从字符串列表中取出元素，并将接收到的内容通过自定义消息发送给主对话框，由主对话框显示。代码如下：

```
01  int CthreadSend::Run()                    //发送线程运行函数
02  {
03      while (bRun)                          //当运行状态为 true 时，执行循环
04      {
05          EnterCriticalSection(&cs);        //进入关键段
06              //判断全局列表中的元素个数是否小于 100
07          if (globalList.GetCount() < 100)
08          {
09              //如果小于 100，则格式化次数和当前时间
10              CTime time = CTime::GetCurrentTime();
11              CString in;
12              in.Format("第%d 次的运行时间=%s", iIndex++,
13                  time.Format( "%Y-%m-%d %H:%M:%S" ));
14                  globalList.AddTail(in);   //将信息添加到列表中
15          }
16          LeaveCriticalSection(&cs);        //离开关键段
17          Sleep(1000);                      //线程延时
18      }
19      return CWinThread::Run();             //返回线程运行函数
20  }
21  int CthreadRecv::Run()                    //接收线程运行函数
22  {
23      while (bRun)                          //当运行状态为 true 时，执行循环
24      {
```

```
25          EnterCriticalSection(&cs);        //进入关键段
26          if (globalList.GetCount() > 0)//判断全局列表中的元素个数是否大于 0
27          {                                 //如果大于 0，则取链表头元素
28              CString out=globalList.RemoveHead();
29              if (hParent)                  //将链表头信息发送到界面中显示
30                  SendMessage(hParent, WM_RECEIVE_MESSAGE,
31              (DWORD)out.GetBuffer(out.GetLength()),out.GetLength());
32          }
33          LeaveCriticalSection(&cs);        //离开关键段
34          Sleep(1000);                      //线程延时
35      }
36      return CWinThread::Run();             //返回线程运行函数
37  }
```

上面的代码定义了接收线程和发送线程的工作函数。CThreadSend 的 Run()函数是发送线程的工作函数，进入临界区后，将当前计数值和当前时间组合成字符串添加到全局变量 globalList 的列表尾，然后退出临界区。CthreadRecv 的 Run()函数是接收线程的工作函数，进入临界区后，从全局变量 globalList 中取出列头元素，并发送消息给主对话框，然后退出临界区。主线程的代码如下：

```
01  void CMTSampleView::OnMenuitemStartrecvthread()//接收按钮的执行函数
02  {
03      if (pThreadRecv)                               //判断接收线程是否有效
04          if (pThreadRecv->bRun==true)
05              return;
06      //创建线程并启动
07      pThreadRecv = (CthreadRecv*)AfxBeginThread
08          (RUNTIME_CLASS(CthreadRecv),NULL);
09      pThreadRecv->hParent = GetSafeHwnd();          //赋值父窗体句柄
10      pThreadRecv->m_bAutoDelete = false;       //赋值自动删除变量为 false
11      if (pThreadRecv)
12          WriteLog("启动接收线程成功");  //输出提示信息
13  }
14  void CMTSampleView::OnMenuitemStartsendthread()//发送按钮的执行函数
15  {
16      if (pThreadSend)     //判断发送线程是否有效
17          if (pThreadSend->bRun==true)
18              return;
19      //创建线程并启动
20      pThreadSend = (CthreadSend*)AfxBeginThread
21          (RUNTIME_CLASS(CthreadSend), NULL);
22      pThreadSend->m_bAutoDelete = false;       //赋值自动删除变量为 false
23      if (pThreadSend)
24          WriteLog("启动发送线程成功");  //输出提示信息
25  }
26  void CMTSampleView::OnMenuitemStopthread() //线程停止按钮处理函数
27  {
28      if (pThreadRecv)
29          pThreadRecv->bRun=false;
30      //赋值接收线程的运行状态变量为 false
31      if (pThreadSend)
32          pThreadSend->bRun=false;
33      //赋值发送线程的运行状态变量为 false
34      WriteLog("停止线程成功");                  //输出提示信息
35  }
```

上面 3 个函数中，OnMenuitemStartrecvthread()函数和 OnMenuitemStartsendthread()函数分别是启动接收线程和启动发送线程，调用 AfxBeginThread()函数启动相应的线程。OnMenuitemStopthread()函数通过是否继续运行的标识变量 bRun 控制线程终止。

```
01  LRESULT CMTSampleView::OnRecvMsg(WPARAM wParam, LPARAM lParam)
02  {
03      CString log;
04      log.Format("接收到消息=%s", wParam);
05      WriteLog(log);
06      return 1;
07  }
```

上面的代码是主对话框对自定义的接收数据消息的处理函数，目前仅是将接收到的数据显示在屏幕上。程序的运行效果如图 23-17 所示。

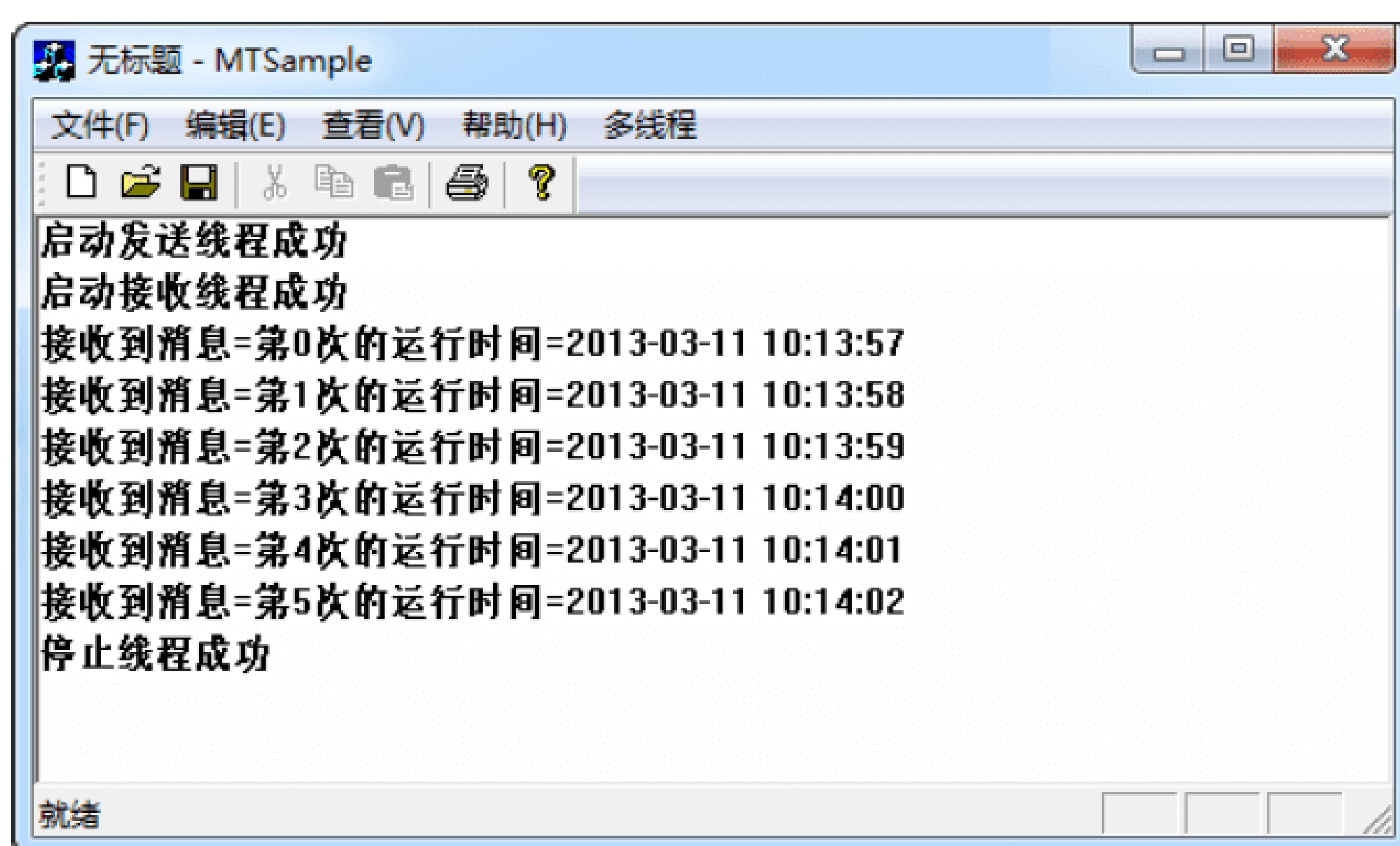


图 23-17　多线程程序实例运行效果

上面这个实例中，用到了前面介绍的使用 MFC 开发多线程技术管理线程，使用全局变量和自定义消息实现线程间的通信，并使用临界区同步对象，实现线程间的同步。虽然实例功能简单，但是其涵盖了多线程编程的各方面技术。读者应该深入理解此实例。

23.7　本 章 小 结

本章介绍了在 Visual Studio 2010 环境下有关多线程程序的开发知识。在引入了多线程工作原理和进程与线程对比的知识后，介绍了多线程程序的开发方法、线程间通信的实现方法以及线程同步的处理，最后以一个实例讲解了如何进行多线程程序的开发。本章的重点是掌握开发多线程程序的方法以及线程同步的控制。本章的难点在于线程同步的处理。第 24 章将开始介绍有关多媒体的操作。

23.8　习　　题

1．通过新线程完成对画图程序的各种操作，主要包括以下操作。

（1）启动画图程序。

（2）关闭画图程序。

（3）启动画图程序后马上将其挂起。

（4）监测画图程序的关闭。

【思路】可以参考 23.3.7～23.3.9 小节所讲的示例完成本题。

2．创建一个控制台项目，通过主线程 main()来创建两个新的线程：ThreadFunc1()和 ThreadFunc2()。详细要求如下。

（1）项目中有一个全局的 int 型的变量 ShareNum，初始赋值为 10。

（2）ThreadFunc1()先访问变量 ShareNum，并将其值修改为 20 后，再输出 ShareNum 的值。

（3）然后 ThreadFunc2()才访问变量 ShareNum，读取其值并输出。

【思路】本题通过全局变量 ShareNum 实现线程的通信。同时还需要同步两个线程，可以使用 23.5 节介绍的 4 种方法中的任意一种来实现。

第 6 篇　多媒体开发

第 24 章　文本字体技术

字体用于在媒体显示器和其他输出设备中绘制文本。Win32API 提供了一组用于安装、选择和查询不同字体的函数。并且提供了字体对象，可以实现有关字体的绘制和一些字体特效。24.1 节中介绍了有关字体对象的基本知识，24.2 节以多个字体效果为例，讲解了在 VC 中如何实现不同的文本字体。

24.1　字体对象

字体对象用于处理有关字体的操作，如字体的粗细、大小和字体的其他样式。本节介绍字体的基本要素、如何创建字体对象和获取字体信息，最后介绍一个字体对象的使用实例。

24.1.1　字体要素

字体是文字特性的集合，包含常用设计的符号。字体的 3 个主要元素是字型、样式和大小。

1．字型

字型是指字体中指定的特征和符号的特性。

2．样式

样式是指字体的重量和是否倾斜。字体的重量可以从淡到黑。下面从轻到重列出了 Win32 字体支持的重量，如表 24-1 所示。

表 24-1　字体重量

支持的重量类型	含　　义	支持的重量类型	含　　义
Thin	细	Semibold	中粗
Extralight	特轻	Bold	粗
Light	轻	Extrabold	特粗
Normal	正常	Heavy	重
Medium	中间		

有 3 种类型的字体是倾斜的：Roman、Oblique 和 Italic。在 Roman 字体中的此特征是直的。在 Oblique 字体中是人工倾斜的。倾斜是通过从 Roman 字体中执行剪切转换实现的。Italic 字体中的此特性是真正的倾斜，并且与设计时的样子相同。

3．大小

字体的大小是一个不精确的值，是指从小写字母 g 的底端到大写字母 M 的顶端的距离，如图 24-1 所示。



图 24-1　字体的大小

指定字体大小的单位为磅。一磅等于一英尺的 0.13837 倍。

24.1.2　创建字体对象

CGdiObject 类提供了多种 Windows 图形设备接口（GDI）对象的基类，如位图、区域、画刷、画笔、调色板和字体。不能直接创建 CGdiObject，而是通过派生类创建对象，如 CPen 或 CBrush，如图 24-2 所示。

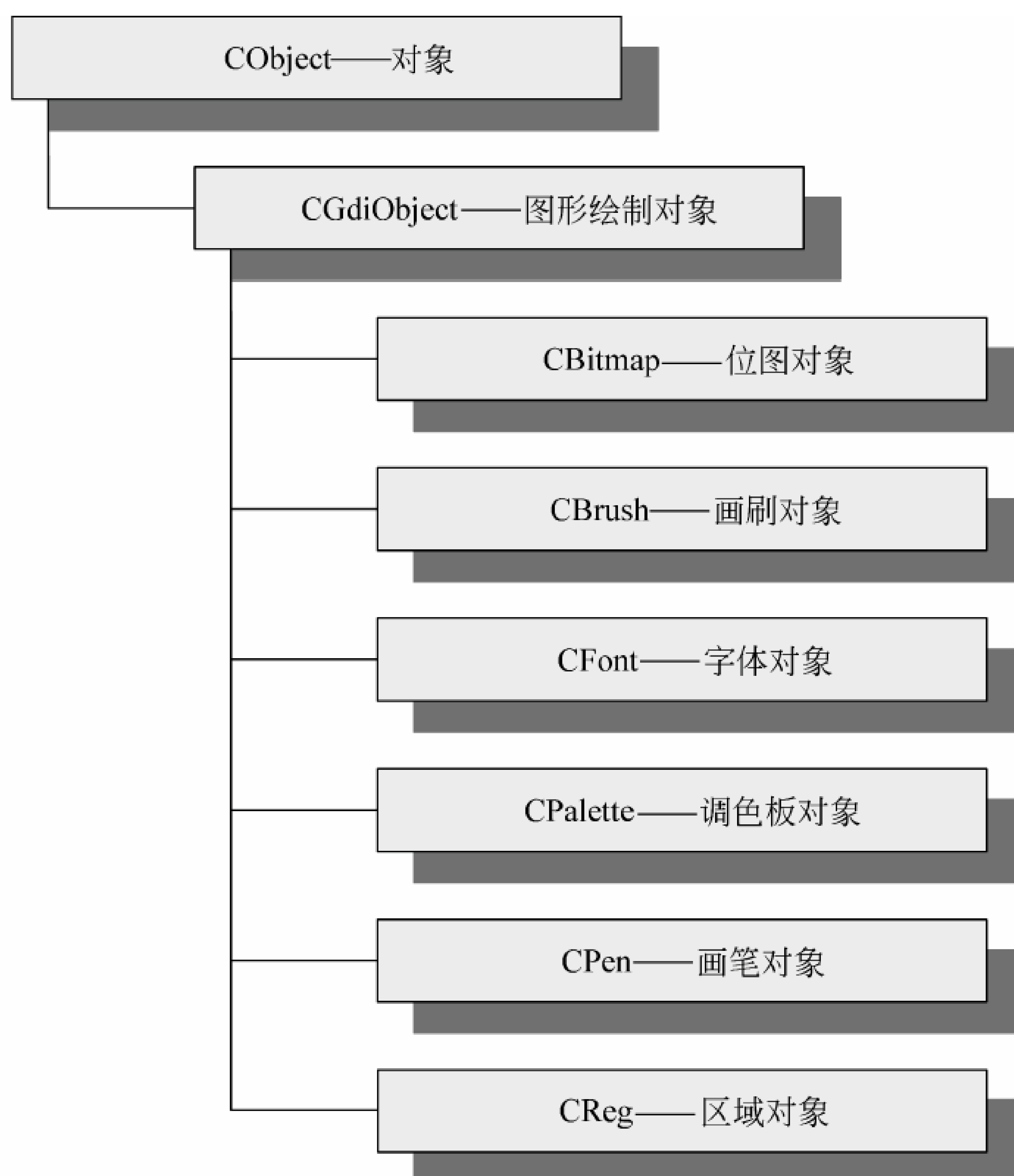


图 24-2　MFC 中图形绘制对象的层次结构

MFC 使用 CFont 类封装了 Windows 图形设备接口字体，并提供了操作字体的成员函数。要使用 CFont 对象，需要构造一个 CFont 对象，使用 CreateFont()函数、CreateFontIndirect()

函数、CreatePointFont()函数或 CreatePointFontIndirect()函数指派一个 Windows 字体，并使用对象的成员函数操作字体。

CreatePointFont()函数和 CreatePointFontIndirect()函数使用起来比 CreateFont 或 CreateFontIndirect()函数更容易，因为其将字体的高度从点大小自动转换成逻辑单位。CFont 类的成员函数如表 24-2 所示。

表 24-2　CFont类的成员函数

成　　员	功　　能
CreateFontIndirect()	使用 LOGFONT 结构中的值初始化 CFont 对象
CreateFont()	使用指定的特征初始化 CFont 对象
CreatePointFont()	使用指定字体类型和字体大小创建字体
CreatePointFontIndirect()	与 CreateFontDirect()功能相同，只是字体高度是使用点的十分之一作为度量单位，而不是逻辑单位
FromHandle()	当给定一个 Windows 的 HFONT 句柄时，返回一个代表 CFont 对象的指针
Operator HFONT()	返回指定给 CFont 对象的 Windows GDI 字体句柄
GetLogFont()	使用附加在 CFont 对象的逻辑字体的信息填充 LOGFONT

24.1.3　获取字体信息

Win32 提供 TEXTMETRIC 结构存放实体字体信息。此函数获取的所有大小值都是逻辑单位，根据当前显示上下文的映射模式确定逻辑单位的大小。其原型为：

```
typedef struct tagTEXTMETRIC { //字体信息结构
    LONG tmHeight;              //指定字体字符的高度
    LONG tmAscent;              //指定字体字符的超出基本线以上的高度
    LONG tmDescent;             //指定字体字符的基本线以下的高度
    LONG tmInternalLeading;     //指定由 tmHeight 成员设置的边界内的空间大小
    LONG tmExternalLeading;     //指定应用程序增加到行之间的空间大小
    LONG tmAveCharWidth;        //指定字体字符宽度的平均值
    LONG tmMaxCharWidth;        //指定字体中最宽字符的宽度
    LONG tmWeight;              //指定字体宽度
    LONG tmOverhang;            //指定增加到字体中的每个字符的外部宽度
    LONG tmDigitizedAspectX;    //指定设计字体的设备水平方向值
    LONG tmDigitizedAspectY;    //指定设计字体的设备垂直方向值
    CHAR tmFirstChar;           //指定字体中定义的第一个字符值
    BCHAR tmLastChar;           //指定字体中定义的最后一个字符值
    BCHAR tmDefaultChar;        //指定字体中用于替换字体中没有定义的字符的字符值
    BCHAR tmBreakChar;          //指定字体中用于定义文本调整的字符值
    BYTE tmItalic;              //指定字体是否为斜体
    BYTE tmUnderlined;          //指定字体是否带有下划线
    BYTE tmStruckOut;           //指定字体是否带有删除线
    BYTE tmPitchAndFamily;      //指定字体种类的信息
    BYTE tmCharSet;             //指定字体的字符集
} TEXTMETRIC;
```

调用 GetTextMetrics()函数可以将当前选择的字体填充到指定的 TEXTMETRIC 结构中，从此结构中可以获取有关字体的信息。

```
BOOL GetTextMetrics(
  HDC hdc,                          //指向设备上下文
  LPTEXTMETRIC lptm );              //指向存放字体信息的 TEXTMETRIC 结构指针
```

如果函数操作成功，则返回非 0（true）；否则返回 0（false）。要获取错误原因，可以调用 GetLastError()函数。

24.1.4　字体对象使用实例

以下代码可以获取字体信息。

```
01  void CFontEffectsSampleView::OnDraw(CDC* pDC)  //视图重绘函数
02  {
03      CFontEffectsSampleDoc* pDoc = GetDocument();//获取文档对象指针
04      ASSERT VALID(pDoc);                         //验证文档对象
05      CRect rc;                                   //定义区域对象
06      GetClientRect(&rc);                         //获取客户区坐标
07      TEXTMETRIC tm;                              //定义字体结构
08      pDC->GetTextMetrics(&tm);                   //获取字体信息
09      pDC->SetTextAlign(TA CENTER | TA TOP);//设置字体对齐方式为上端居中
10      CString strText;                            //字体信息字符串
11      strText.Format("字体高度=%d   字体宽度=%d   斜体=%d
12          下划线=%d   删除线=%d",tm.tmHeight, tm.tmWeight,
13          tm.tmItalic, tm.tmUnderlined, tm.tmStruckOut);
14      pDC->TextOut((rc.left + rc.right) / 2, 0, strText);
15  }
```

上面代码使用 GetClientRect()函数获取文档区域，使用 CDC 类的 GetTextMetrics()函数获取字体信息，并使用 TextOut()函数在文档上显示出来。程序的运行效果如图 24-3 所示。

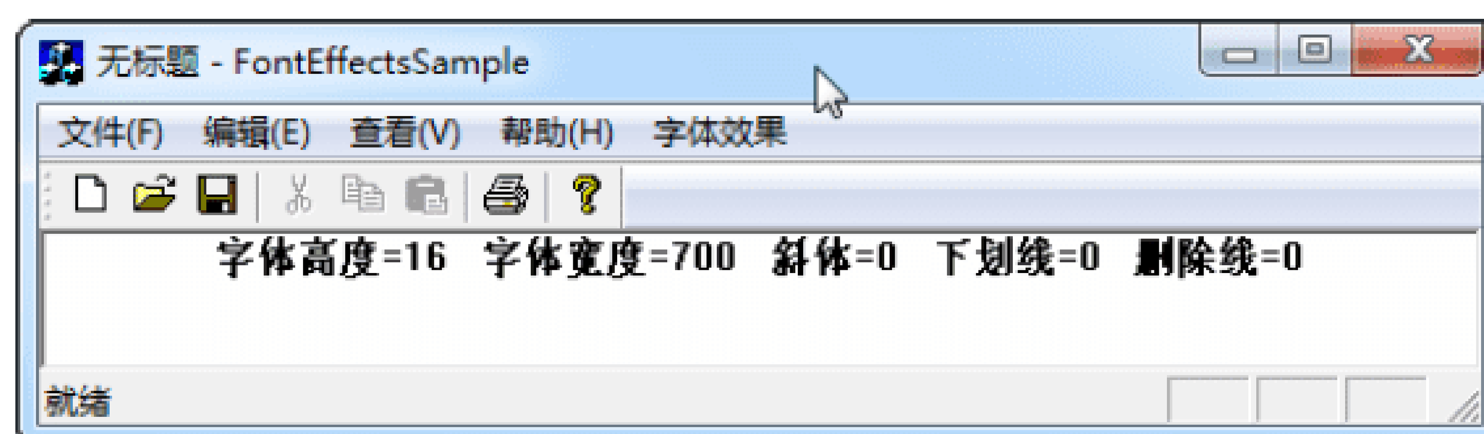


图 24-3　字体信息获取程序运行效果

24.2　字 体 效 果

本节在 24.1 节的基础上，介绍几种常见字体效果的实现。在介绍这些字体时，会反复使用一些处理函数。在学习本节内容时，需要融会贯通地了解这些函数的使用方法，这样就可以触类旁通地实现许多字体效果。

24.2.1　如何设计空心字

在 VC 中可以设计空心字，设计思路是使用 CFont 字体对象设置要显示的字体的样式，

使用 CPen 画笔对象设置显示文字时使用的画笔样式，设置好后将文字在文档界面上显示出来，然后绘制空心字的效果。具体代码如下：

```
01  void CFontEffectsSampleView::OnMenuKongxinFont()//绘制空心字
02  {
03      CClientDC dc(this);                  //获得对话框的客户区设备上下文句柄
04      LOGFONT lf;                          //更改当前字体
05      dc.GetCurrentFont()->GetLogFont(&lf);
06      CFont font;                          //保存设备上下文最初使用的字体对象
07      lf.lfCharSet=134;
08      lf.lfHeight=-80;
09      lf.lfWidth=0;
10      strcpy(lf.lfFaceName, "宋体");
11      font.CreateFontIndirect(&lf);                     //创建字体
12      CFont *pOldFont=dc.SelectObject(&font);           //装载字体
13      dc.SetBkMode(TRANSPARENT);                        //设置字体的模式是透明
14      CPen pen(PS_SOLID, 2, RGB(255, 50, 0));           //更改当前画笔
15      CPen *pOldPen=dc.SelectObject(&pen);              //装载画笔
16      dc.BeginPath();                                   //开始绘制
17      dc.TextOut(10, 10, "这里是空心字效果示例");
18      dc.EndPath();                                     //结束绘制
19      dc.StrokePath();
20      dc.SelectObject(pOldFont);             //恢复设备上下文的字体原有设置
21      dc.SelectObject(pOldPen);                         //恢复画笔
22  }
```

上面代码使用 GetCurrentFont()函数获得当前使用的字体。对返回的字体对象进行修改，使用修改后的 CPen 对象绘制文字。要注意的是，绘制完成后，需要重新恢复原来的上下文设置。程序运行的效果如图 24-4 所示。

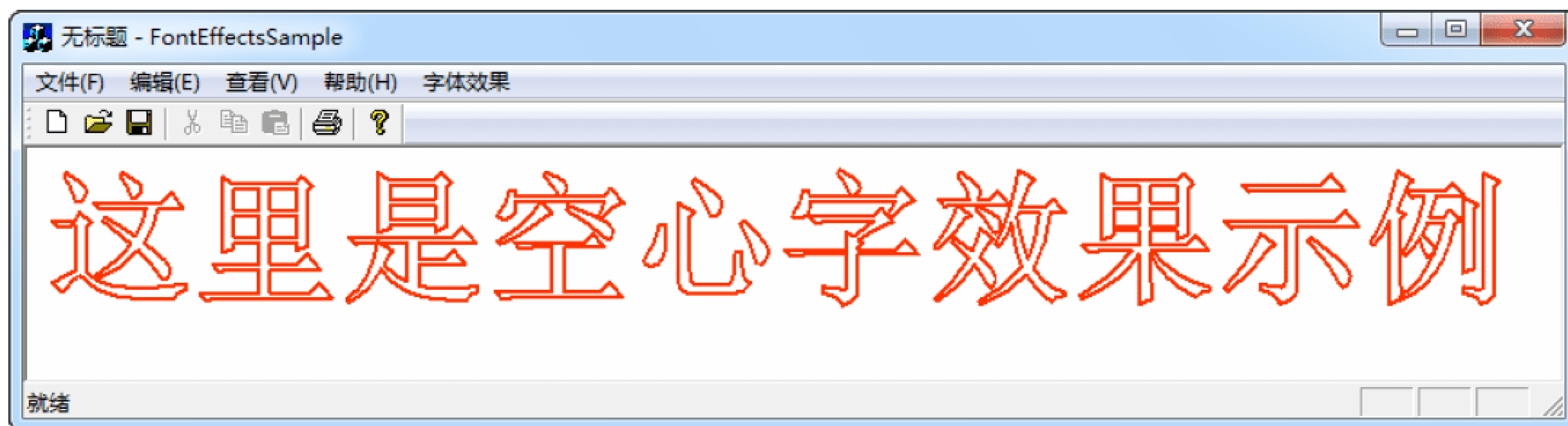


图 24-4　空心字效果

24.2.2　渐变颜色的字体

在 VC 中可以设计颜色渐变的字体，设计思路是使用 CFont 字体对象设置要显示的字体的样式，使用 CPen 画笔对象设置显示文字时使用的画笔样式，使用 CBrush 画刷设置要实现颜色渐变的参数。设置后，将文字在文档界面上显示出来，然后使用画刷绘制颜色渐变的效果。具体代码如下：

```
01  void CFontEffectsSampleView::OnMenuitemJianbianFont()//渐变字体的实现
02  {
```

```
03      CClientDC dc(this);                    //获得对话框的客户区设备上下文句柄
04      LOGFONT lf;                            //更改当前字体
05      dc.GetCurrentFont()->GetLogFont(&lf);              //获取字体
06      CFont font, *pOldFont;                             //定义原来的字体变量
07      lf.lfCharSet=134;
08      lf.lfHeight=-50;
09      lf.lfWidth=0;
10      strcpy(lf.lfFaceName, "宋体");
11      font.CreateFontIndirect(&lf);                      //创建字体
12      pOldFont=dc.SelectObject(&font);                   //选择字体
13      dc.SetBkMode(TRANSPARENT);                         //设置字体是透明的
14      CPen pen(PS_NULL, 1, RGB(0, 0, 255));              //更改当前画笔为空
15      CPen *pOldPen=dc.SelectObject(&pen);               //选择画笔
16      CBrush br, *pOldBrush =dc.SelectObject(&br);       //选择画刷
17      dc.BeginPath();                                    //开始一个路径
18      dc.TextOut(10, 10, "这里是渐变颜色字体示例");
19      dc.EndPath();
20      dc.SelectClipPath(RGN_COPY);                       //绘制渐变效果
21      for (int i=255; i>0; i--)
22      {
23          int iRadius=(800*i)/255;
24          dc.SelectObject(pOldBrush);
25          br.DeleteObject();
26          br.CreateSolidBrush(RGB(i, 192, 192));
27          dc.SelectObject(&br);
28          dc.Ellipse(-iRadius, -iRadius/3, iRadius, iRadius/3);
29      }
30      //恢复设备上下文的原有设置
31      dc.SelectObject(pOldFont);
32      dc.SelectObject(pOldPen);
33      dc.SelectObject(pOldBrush);
34  }
```

上面代码使用 GetCurrentFont()函数获得当前使用的字体，对返回的字体对象进行修改。其中，for 循环语句实现了渐变的效果。要注意的是，绘制完成后，需要重新恢复原来的上下文设置。程序运行的效果如图 24-5 所示。

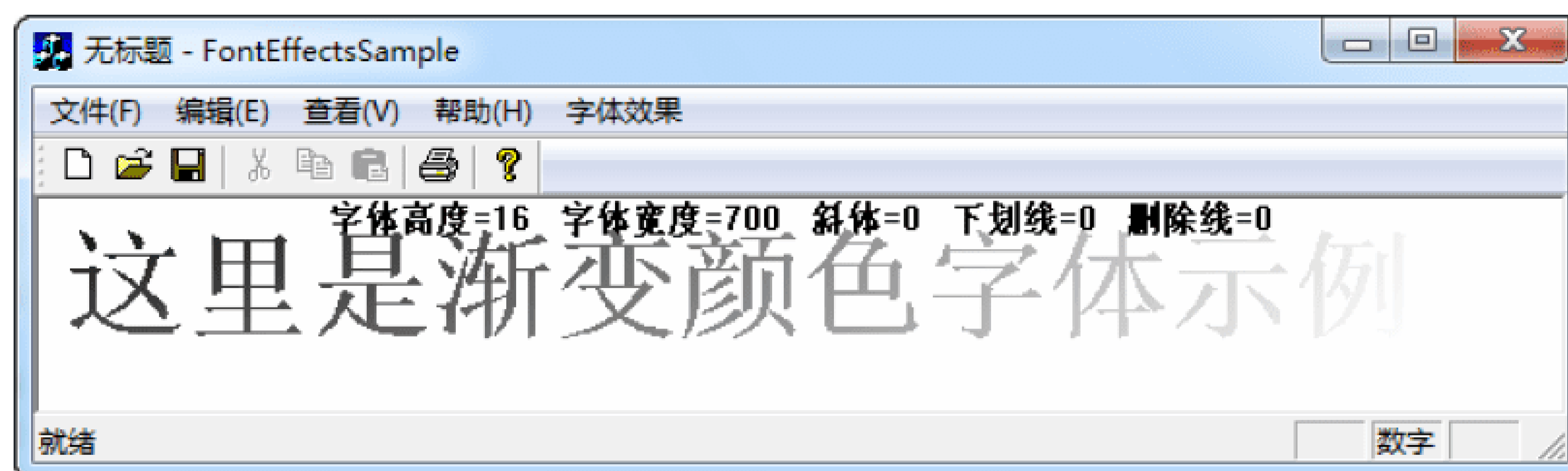


图 24-5　渐变颜色字体效果

24.2.3　获取路径信息点

GetPath()函数可以获取 Windows GDI 路径中的所有信息点。使用这些信息点可以绘制直线和曲线等。其函数原型为：

```
int GetPath(
  HDC hdc,              //包含关闭路径的设备上下文的句柄
  LPPOINT lpPoints,     //表示存放线的结束点和曲线控制点的 POINT 结构的点数组的指针
  LPBYTE lpTypes,       //表示放置在字节数组中的指针
  int nSize);           //指定 lpPoints 中存放的 POINT 结构的数目
```

其中，lpTypes 参数表示放置在字节数组中的指针，取值可以是如下几种。

- PT_MOVETO：表示 lpPoints 参数中存放的是连接点的起点。
- PT_LINETO：表示 lpPoints 参数中存放的是线的结束点。
- PT_BEZIERTO：表示 lpPoints 参数中存放的是控制点或曲线的结束点。

nSize 参数指定 lpPoints 中存放的 POINT 结构的数目。如果 nSize 参数为非 0 值，则返回值为枚举点的数目；如果 nSize 为 0，则返回值为路径中的所有点的数目。

24.2.4　文字跟随鼠标

在 VC 中可以实现文字跟随鼠标的效果。实现这个效果需要使用 CClientDC 的 TextOut()函数，在鼠标移动到的位置显示要显示的文字。使用定时器，每次将位置增加定值。具体代码如下：

```
01  void CFontEffectsSampleView::OnMouseMove(UINT nFlags, CPoint point)
02  {
03      mousePoint.x = point.x;                        //保存鼠标点 x 值
04      mousePoint.y = point.y;                        //保存鼠标点 y 值
05      Invalidate();
06      CView::OnMouseMove(nFlags, point);             //调用基类函数
07  }
08  void CFontEffectsSampleView::OnPaint()             //绘制函数
09  {
10      CPaintDC dc(this);                             //在设备上下文中绘制
11      dc.TextOut(mousePoint.x+3, mousePoint.y+3, "欢迎"); //输出信息
12  }
```

在上面代码中，dc.TextOut 语句的第一个和第二个参数指定了要显示文字的位置，之所以要将传入的点的位置加上 3，是因为鼠标有个箭头，需要向外一点，以便完整显示。此例中显示的鼠标文字为“欢迎”。程序运行效果如图 24-6 所示。

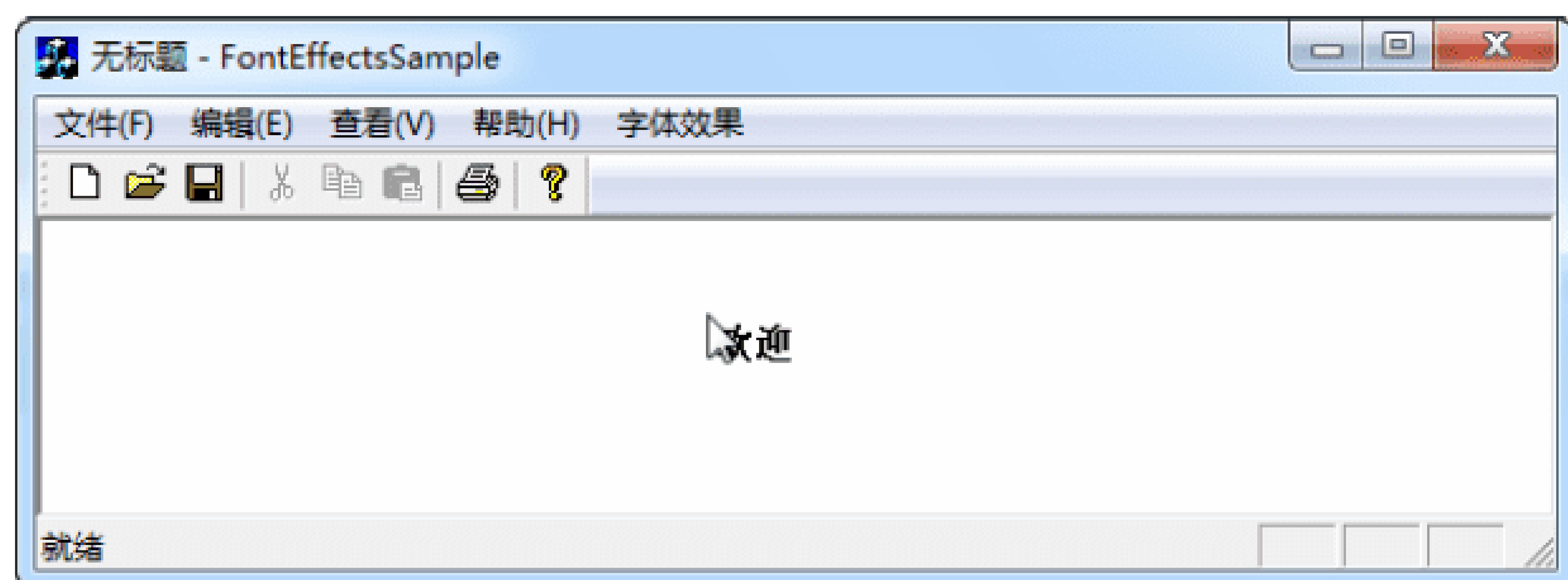


图 24-6　文字跟随鼠标的运行效果

24.2.5　如何实现旋转字体

在 VC 中可以设计旋转字体，设计思路是使用 LOGFONT 结构定义新的字体。其中，

lfEscapement 成员表示字体倾斜的角度，以十分之一度为单位倾斜，设置好后将文字在文档界面上显示出来。具体代码如下：

```
01  void CFontEffectsSampleView::OnMenuitemRotateFont()//实现旋转字体
02  {
03      CClientDC dc(this);                    //获得对话框的客户区设备上下文句柄
04      LOGFONT lf;                            //定义字体属性
05      lf.lfHeight = 50;
06      lf.lfWidth = 0;
07      lf.lfEscapement = 500;                 //倾斜 30°，十分之一度为单位
08      lf.lfOrientation = 0;
09      lf.lfItalic = false;
10      lf.lfUnderline = false;
11      lf.lfStrikeOut = false;
12      lf.lfCharSet = GB2312 CHARSET;
13      strcpy(lf.lfFaceName,"楷书");
14      CFont font;                                    //创建字体
15      font.CreateFontIndirect(&lf);
16      CFont *pOldFont  = dc.SelectObject(&font);   //更改当前字体
17      dc.SetBkMode(TRANSPARENT);                     //绘制字体
18      dc.SetTextColor(RGB(0,255,0));                 //设置字体颜色
19      dc.TextOut(10,300,"这里是旋转字体示例");       //输出内容
20      dc.SelectObject(pOldFont);                 //恢复设备上下文的原有设置
21  }
```

在上面代码中，LOGFONT 结构的 lfHeight 成员定义字体的高度，lfWidth 成员定义字体的宽度，lfEscapement 成员定义字体的倾斜角度，lfItalic 成员定义字体是否倾斜，lfUnderline 成员定义字体是否有下划线，lfStrikeOut 成员定义字体是否有删除线，lfCharSet 成员定义字符集。定义好这些属性，使用 CreateFontIndirect()函数创建新字体，然后使用 SelectObject()函数装载字体，使用 TextOut 显示出旋转字体。运行效果如图 24-7 所示。

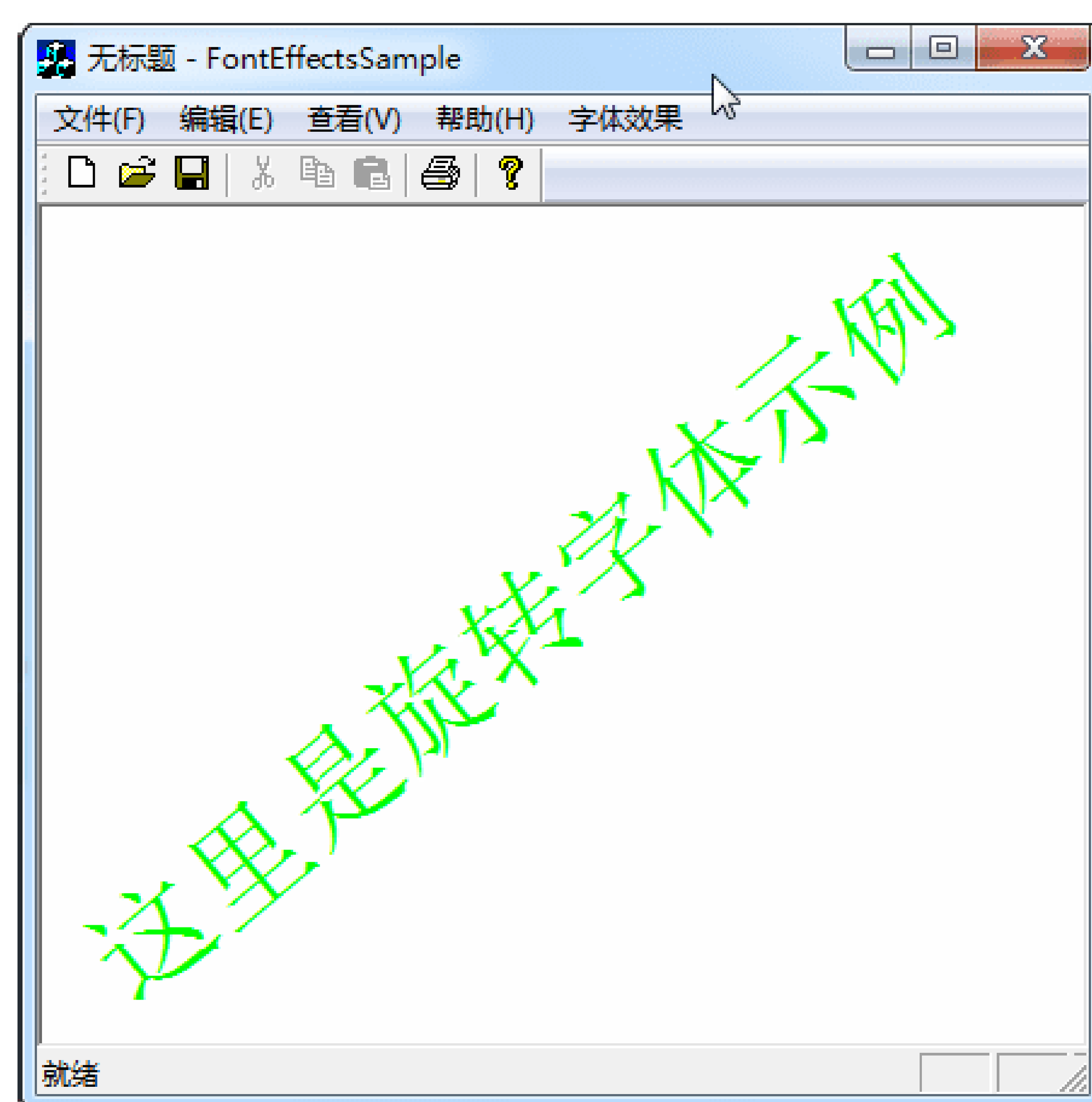


图 24-7　旋转字体效果

24.2.6　文字水平滚动

在 VC 中可以实现文字的水平滚动效果。实现这个效果需要使用定时器，每次将位置

增加定值，并调用 Invalidate()函数使得桌面重绘。在重绘函数中输出文字。具体代码如下：

```
01  void CFontEffectsSampleView::OnTimer(UINT nIDEvent)//定时器处理函数
02  {
03      CRect   rect;                                       //定义区域变量
04      GetClientRect(rect);                                //获取客户工作区
05      int  iWidth=rect.Width();                           //获取区域宽度
06      Invalidate();
07      UpdateWindow();                                     //更新窗口显示
08      m_iExtend+=2;                                       //增加水平偏移量
09      if (m iExtend > iWidth)
10          m iExtend = 0;
11      CView::OnTimer(nIDEvent);                       //调用基类的定时器函数
12  }
13  void CFontEffectsSampleView::OnPaint()              //绘制函数
14  {
15      CPaintDC dc(this);
16      dc.TextOut(m_iExtend,10,"水平文字滚动测试");    //在偏移量处绘制函数
17  }
```

在上面代码中，OnTimer()函数是定时器处理函数，每次将位置增加 2 个元素。调用 Invalidate()函数进行刷新，则系统会调用 OnPaint()函数，在此函数中将要滚动的文字输出。程序运行效果如图 24-8 所示。

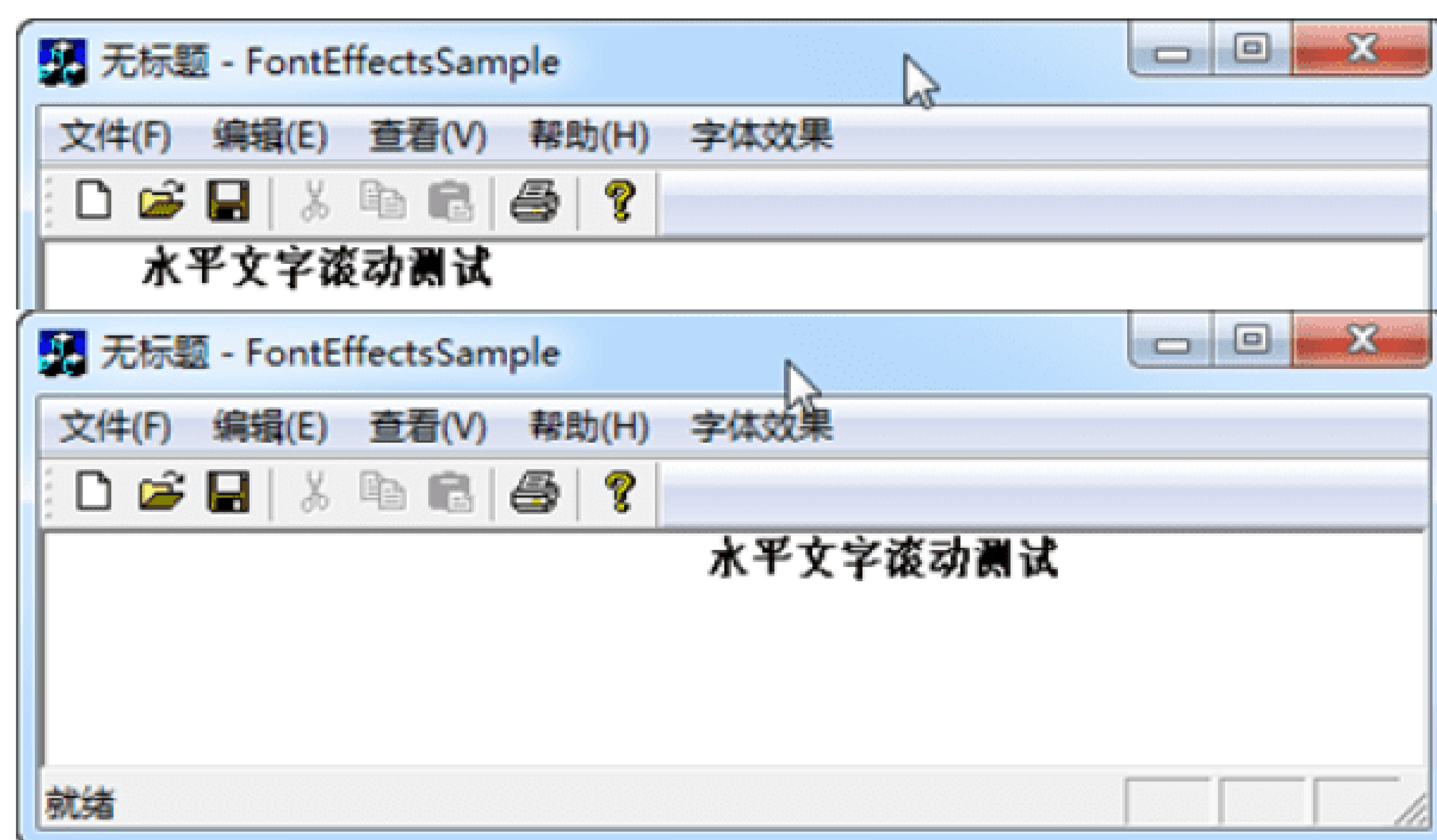


图 24-8　水平文字滚动效果

24.2.7　字体垂直滚动

在 VC 中可以实现文字的垂直滚动效果。实现这个效果需要使用定时器，每次将垂直位置增加定值，并调用 Invalidate()函数使得桌面重绘。在重绘函数中输出文字。具体代码如下：

```
01  void CFontEffectsSampleView::OnTimer(UINT nIDEvent)//定时器处理函数
02  {
03      CRect   rect;                                       //定义区域变量
04      GetClientRect(rect);                                //获取客户工作区
05      int iHeight = rect.Height();                        //获取区域高度
06      Invalidate();
07      UpdateWindow();                                     //更新窗口显示
08      m_iYExtend+=2;                                      //增加垂直偏移量
```

```
09      if (m_iYExtend > iHeight)
10          m_iYExtend = 0;
11      CView::OnTimer(nIDEvent);                          //调用基类的定时器函数
12  }
13  void CFontEffectsSampleView::OnPaint()                          //绘制函数
14  {
15      CPaintDC dc(this);
16      //在偏移量处绘制
17      CRect rc(m_iXExtend, m_iYExtend, m_iXExtend+15, m_iYExtend+150);
18      dc.DrawText("垂直文字滚动测试", &rc,
19          DT_EDITCONTROL|DT_WORDBREAK|DT_CENTER);
20  }
```

在上面代码中，OnTimer()函数是定时器处理函数，每次将垂直位置增加 2 个元素。调用 Invalidate()函数进行刷新，则系统会调用 OnPaint()函数，在此函数中将要滚动的文字输出。程序运行效果如图 24-9 所示。

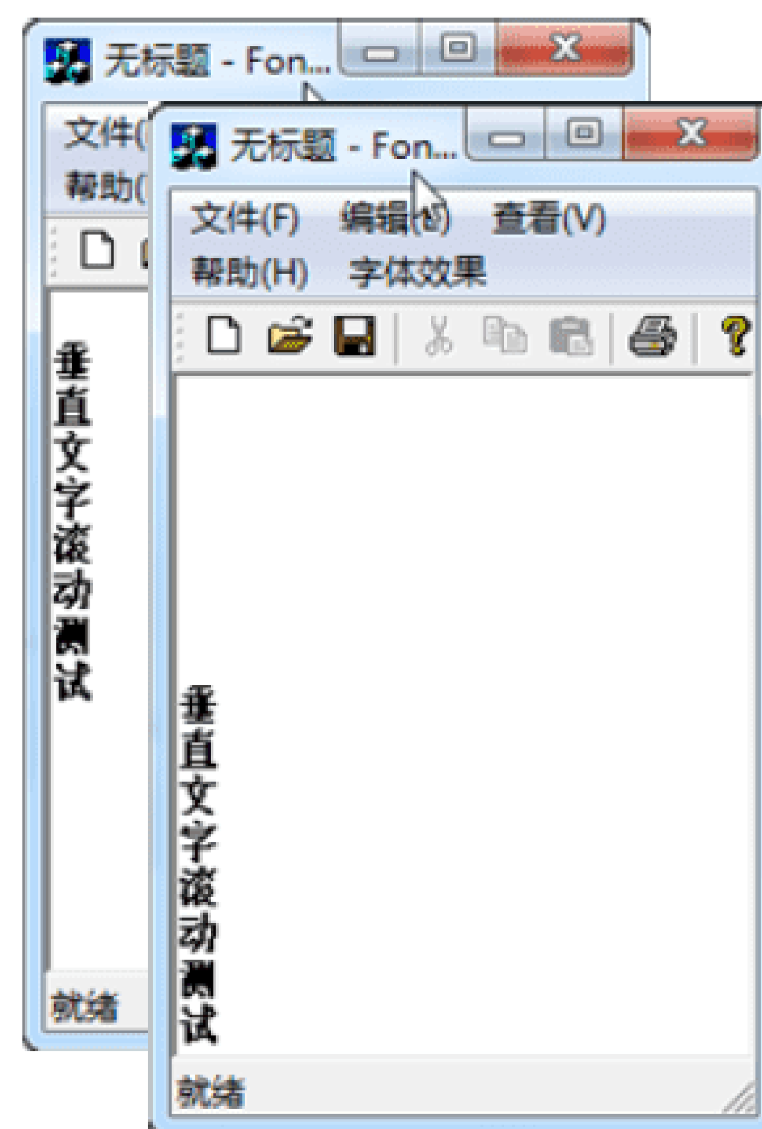


图 24-9　垂直滚动文字效果

24.2.8　设计 3D 立体文字

在 VC 中可以设计 3D 立体字，设计思路是创建使用的字体对象，并使用 SetTextColor()函数分别进行高亮状态显示和阴影状态显示，这样组合起来的效果就是 3D 效果。具体代码如下所示。

```
01  void CFontEffectsSampleView::OnMenuitem3dFont()             //3D 字体效果
02  {
03      CClientDC dc(this);                  //获得对话框的客户区设备上下文句柄
04      LOGFONT lf;                                  //定义字体属性
05      lf.lfHeight = 50;
06      lf.lfWidth = 0;
07      lf.lfEscapement = 0;
08      lf.lfOrientation = 0;
09      lf.lfWeight = FW_HEAVY;
10      lf.lfItalic = false;
11      lf.lfUnderline = false;
12      lf.lfStrikeOut = false;
13      lf.lfCharSet = GB2312_CHARSET;
14      strcpy(lf.lfFaceName,"隶书");
15      CFont font;                                             //创建字体
16      font.CreateFontIndirect(&lf);
17      CFont *pOldFont  = dc.SelectObject(&font);              //更改当前字体
18      dc.SetBkMode(TRANSPARENT);                              //绘制字体
19      dc.SetTextColor(::GetSysColor(COLOR_3DDKSHADOW));
20      CString text = "这里是 3D 字体示例";
21      CRect rc;
22      GetClientRect(&rc);                                     //获取工作区域
23      dc.BeginPath();                                         //开始一个路径
24      //绘制内容
25      dc.DrawText(text,rc,DT_SINGLELINE|DT_LEFT|DT_VCENTER|DT_CENTER);
26      dc.SetTextColor(::GetSysColor( COLOR_3DHILIGHT) ); //设置文本颜色
27      //绘制阴影文本内容
28      dc.DrawText( text, rc+CPoint(2, 2),
```

```
29          DT_SINGLELINE|DT_LEFT|DT_VCENTER|DT_CENTER);
30      dc.EndPath();
31      dc.StrokePath();                                            //绘制路径
32      dc.SelectObject(pOldFont);                                  //恢复字体
33  }
```

上面代码使用 SetTextColor()函数设置字体颜色。其中，COLOR_3DDKSHADOW 参数表示以阴影状态显示，COLOR_3DHILIGHT 参数表示以高亮状态显示，使用 DrawText()函数显示文本内容。程序运行的效果如图 24-10 所示。

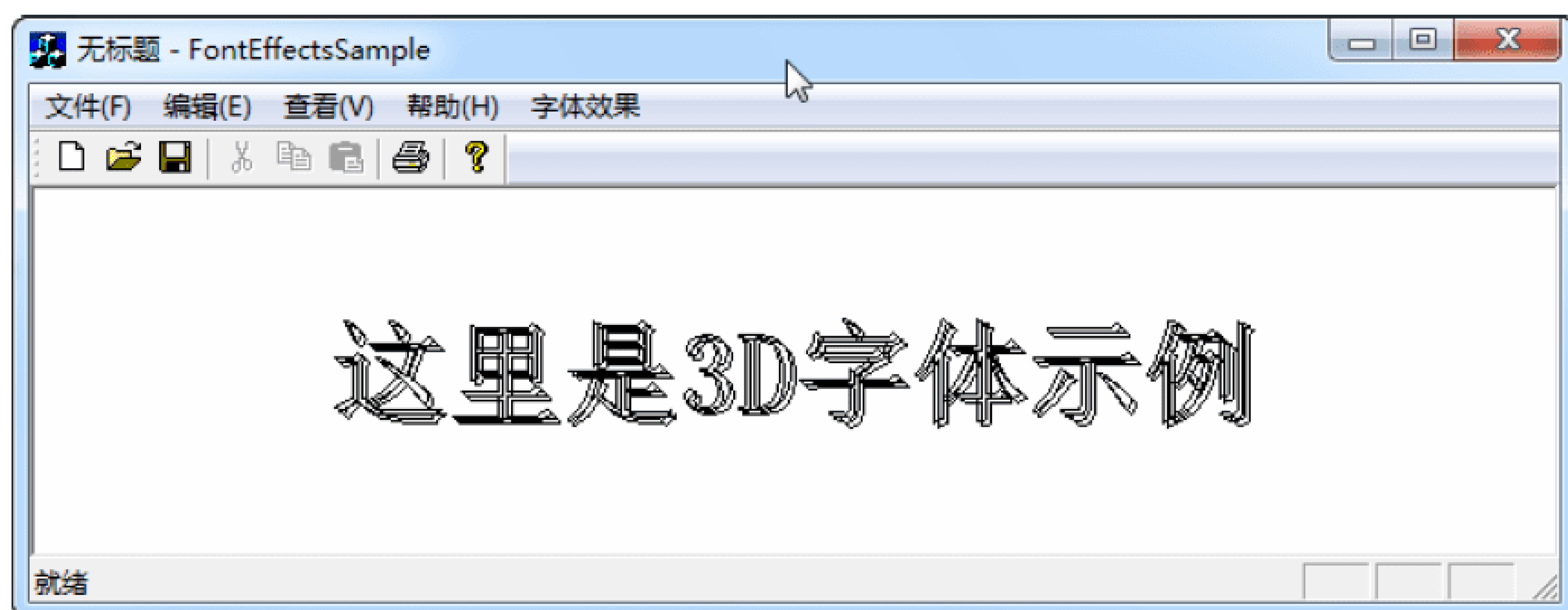


图 24-10　3D 字体示例

24.3　本 章 小 结

本章主要介绍了有关文字字体的操作。本章重点介绍了字体的创建和信息获取以及字体特效的实现。本章难点在于编写各种特效字体，如空心字、颜色渐变的字体实现、文字跟随鼠标移动、旋转文字、3D 文字的设计、水平滚动文字和垂直滚动文字。通过本章的学习，读者应该掌握字体对象的使用方法，达到举一反三的目的。第 25 章将介绍有关图形和图像的开发。

24.4　习　　题

创建基于单文档的应用程序，对字符串“习题测试”完成以下操作：

（1）设置字体的宽为 50 像素，并使用楷体字体。

（2）将字符串逆时针旋转 60° 后显示在客户区。

（3）水平滚动旋转后的字符串。

第 25 章　图形与图像编程

早期 Windows 提供了 GDI（Graphics Device Interface）图形设备接口，它为应用程序操作图形设备提供了统一的访问接口，适用于所有的 Windows 操作系统。随着对图形处理的要求越来越高，Windows 提供了继承自 GDI 的 GDI+技术，在 GDI 基础上，重新调整了接口，使得在 Windows 平台下进行图形和图像开发更加方便。本章将结合 GDI 和 GDI+技术，讲述 Visual Studio 2010 环境下图形与图像的程序开发。

25.1　位图和区域对象

位图是用于创建、操作和存储图像为磁盘文件的图形对象。区域对象用于操作各种类型的区域。位图对象和区域对象是在 Windows 环境下进行图形处理的基础。本节将讨论位图和区域的操作。

25.1.1　设备相关位图（DDB）

DDB（Device-Dependent Bitmaps）即设备相关位图，使用 BITMAP 结构表示。此结构的成员使用像素指定矩形区域的宽度和高度，宽度数组从设备调色板转换成像素，并通过颜色面板和颜色位值指定设备的颜色格式。程序可以通过 GetDeviceCaps()函数和相应的常数获取设备的颜色格式。

设备相关位图分为两种：可废弃的 DDB 和不可废弃的 DDB。如果位图没有载入设备上下文和系统内存比较低，系统会丢弃可废弃的设备相关位图。使用 CreateDiscardableBitmap()函数可以创建可废弃设备相关位图。使用 CreateBitmap()函数、CreateCompatibleBitmap()函数或 CreateBitmapIndirect()函数可以创建不可废弃的设备相关位图。

应用程序可以通过初始化需要的结构和 CreateDIBitmap()函数，从设备无关位图创建设备相关位图。在调用 CreateDIBitmap 中指定 CBM_INIT 与调用 CreateCompatibleBitmap()函数创建设备格式的设备相关位图是相同的，然后调用 SetDIBits()函数将设备无关位图的位值转换成设备相关位图。调用 GetDeviceCaps()函数指定 RASTERCAPS 的标记值为 RC_DI_BITMAP，可以判断设备是否支持 SetDIBits()函数。

描述设备相关位图的 BITMAP 结构可以定义位图类型、宽度、高度、颜色格式和位图的位值，其定义如下：

```
typedef struct  tagBITMAP {
  LONG bmType;                              //指定位图的类型，此成员必须为 0
  LONG bmWidth;                             //指定位图的宽度，单位是像素
  LONG bmHeight;                            //指定位图的高度，单位是像素
  LONG   bmWidthBytes;                      //指定每个扫描行的字节数目
```

```
    WORD   bmPlanes;                          //指定颜色面板的个数
    WORD   bmBitsPixel;                       //指定表示像素的颜色需要的位数
    LPVOID bmBits;          //指定位图位值的位值指针，此成员必须是字符数组的长指针
} BITMAP;
```

位图格式使用黑白色和彩色。黑白色位图使用单位单面板格式。每个扫描符占用 32 位。黑白色设备上的像素是黑色或白色。如果位图上相应位的值为 1，则此像素被设置为前景颜色。如果位图上相应位的值为 0，则此像素被设置为背景颜色。所有具有 RC_BITBLT 设备能力的设备都支持位图。每个设备具有唯一的颜色格式。使用 GetDIBits()函数和 SetDIBits()函数可以从一个设备转换位图到另一个设备上。

25.1.2　CBitmap 应用实例

CBitmap 类封装了 Windows 图形设备接口位图，并提供了操作位图的成员函数。使用 CBitmap 对象构造对象，使用初始化成员函数将位图句柄附加到位图对象上，然后调用对象的成员函数。使用 CBitmap 类的 LoadBitmap()函数可以装载指定名称的资源。以下显示了装载 IDB_BITMAP_FACE 位图资源，并在整个画布上显示位图的代码。

```
01  void CGDISampleView::OnMenuitemBitmap()    //绘制位图
02  {
03      CBitmap bitmap;                         //定义 CBitmap 变量
04      CDC *pDC = GetDC();                     //获取 CDC
05      bitmap.LoadBitmap(IDB_BITMAP_FACE);    //装载位图
06      CDC memDC;                              //获取内存上下文
07      memDC.CreateCompatibleDC(pDC);       //创建与 pDC 兼容的内存设备上下文
08      memDC.SelectObject(&bitmap);            //选择位图对象
09      BITMAP bmp;                             //定义 BITMAP 对象
10      bitmap.GetBitmap(&bmp);                 //装载位图
11      CRect rect;                             //获取工作区大小
12      GetClientRect(&rect);
13      //在设备上下文绘制位图
14      pDC->StretchBlt(0, 0, rect.Width(), rect.Height(), &memDC,
15          0, 0, bmp.bmWidth, bmp.bmHeight, SRCCOPY);
16      memDC.DeleteDC();                       //删除设备上下文
17      ::DeleteObject(&bitmap);                //删除位图对象
18  }
```

上面代码首先使用 CBitmap 类的 LoadBitmap()成员函数装载了 IDB_BITMAP_FACE 位图资源，并使用 CDC 类的 StretchBlt()函数将位图显示在画布上。有关 CDC 类的使用后面会陆续讲解。此程序运行的效果如图 25-1 所示。

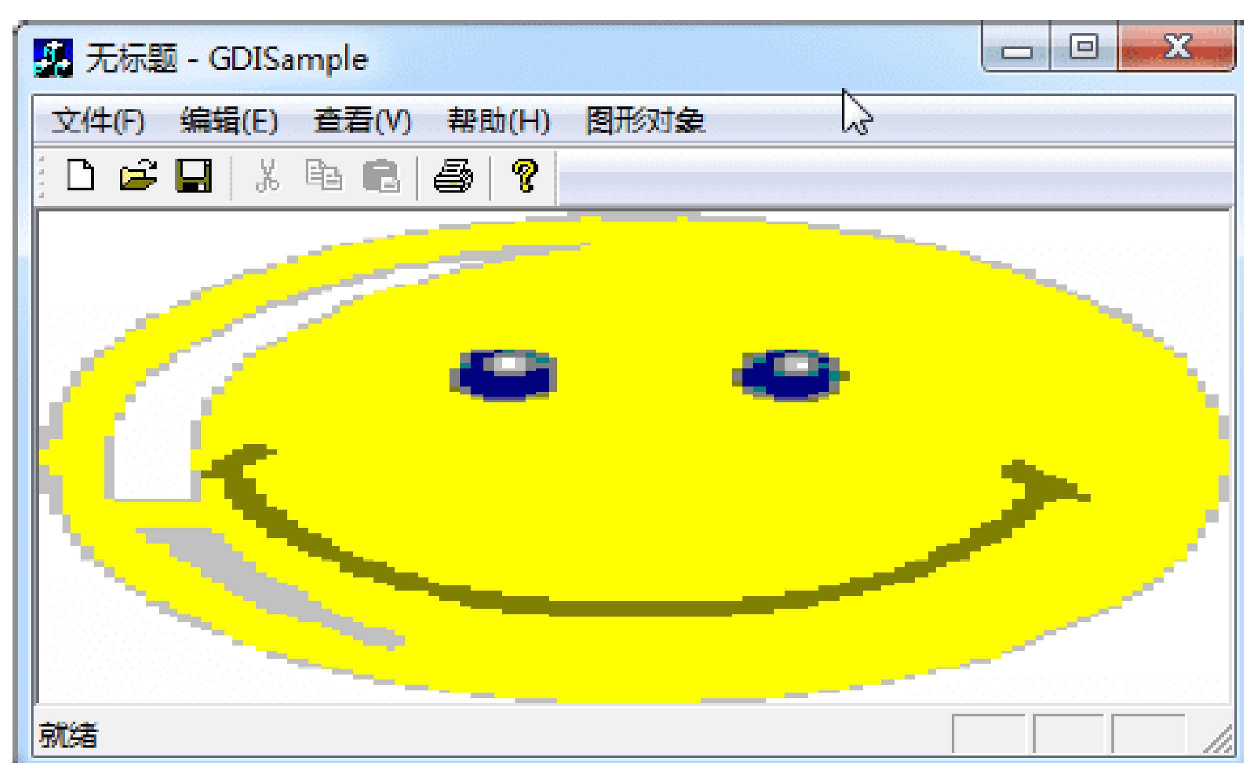


图 25-1　显示位图的程序运行效果

25.1.3　设备无关位图（DIB）

DIB（Device-Independent Bitmaps）即设备无关位图，是包含一个与设备无关的颜色表的位图。颜色表描述像素值如何与 RGB 颜色值对应。RGB 是通过光描述颜色的模型，表示颜色占有的红色（Red）、绿色（Green）和蓝色（Blue）的比例。一个设备无关位图包含下列颜色和大小信息。

- 创建矩形图像的设备的颜色格式。
- 创建矩形图像的设备的方法。
- 创建矩形图像的设备的调色板。
- 映射矩形图像的 RGB 三色像素数组。
- 如果对图像进行压缩，还包含数据压缩方案。

这些颜色和大小信息存储在 BITMAPINFO 结构中。其中包含信息头结构和两个或多个 RGBQUAD 结构。位图信息头结构指定矩形像素的大小、描述设备颜色的技术和用于减小位图大小的压缩方案标识符。RGBQUAD 结构表示显示像素矩形的颜色。设备无关位图分为以下两种。

- “从下到上”的与设备无关位图，起点位于左下角。
- “从上到下”的与设备无关位图，起点位于左上角。

在位图信息头结构中的 Height 成员指定的 DIB 高度，如果是正值，则位图是“从下到上”的设备无关位图；如果高度是负值，则是“从上到下”的设备无关位图，从上到下的 DIB 不能压缩。

颜色格式指定颜色面板数目和颜色位值。位图的颜色面板数总是 1，对于单色位图，颜色位值的数目也是 1；对于 VGA 位图，颜色位值的数目是 4；在其他颜色设备上位图的颜色位值是 8、16、24 或 32。要获取特殊设备，如打印机的颜色位值，可以通过调用 GetDeviceCaps()函数，在第二个参数中指定其值为 BITSPIXEL 进行获取。

25.1.4　区域对象（CRgn）

CRgn 类封装了 Windows 区域图形设备接口。区域是对话框中的椭圆形或多边形区域。通过 CDC 类的裁剪成员函数和 CRgn 类的成员函数使用区域。CRgn 的成员函数创建、修改和获取有关区域对象的信息。使用 CRgn 可以创建矩形、椭圆形、多边形和圆形。其函数原型为：

```
BOOL CreateRectRgn( int x1, int y1, int x2, int y2 );
BOOL CreateEllipticRgn( int x1, int y1, int x2, int y2 );
BOOL CreatePolygonRgn( LPPOINT lpPoints, int nCount, int nMode );
BOOL CreatePolyPolygonRgn(LPPOINT lpPoints,LPINT lpPolyCounts,
                          int nCount,int nPolyFillMode);
BOOL CreateRoundRectRgn( int x1, int y1, int x2, int y2, int x3, int y3 );
```

上面的函数中，CreateRectRgn()函数创建矩形、CreateEllipticRgn()函数创建椭圆形、CreatePolygonRgn()函数创建多边形、CreatePolyPolygonRgn()函数创建多个多边形、CreateRoundRectRgn 创建圆角矩形。其中，x1、y1、x2 和 y2 这 4 个坐标值指定矩形或包

含椭圆形的矩形的定点或圆角矩形的左上角和右下角的坐标。lpPoints 参数表示创建多边形的各个顶点的集合。nCount 参数表示坐标点的个数。nMode 参数和 nPolyFillMode 参数表示图形填充方法。lpPolyCounts 参数表示要创建的多边形的个数。25.1.5 小节将结合一个实例讲解 CRgn 实例的应用。

25.1.5　CRgn 应用实例

本小节讲解使用 CRgn 对象创建一个椭圆形对话框的方法。使用同样的方法也可以创建圆形或圆角矩形的对话框。首先创建区域对象，使用区域对象的创建形状区域的方法创建对话框区域，本小节使用 CreateEllipticRgn()方法创建椭圆形区域。最后调用 SetWindow-Rgn()函数设置对话框的区域。这些工作需要在 OnInitDialog()对话框初始函数中调用，代码如下：

```
01  BOOL CDlgEllip::OnInitDialog()                         //绘制椭圆形对话框
02  {
03      CDialog::OnInitDialog();                           //调用基类对话框
04      SetWindowText(_T("椭圆对话框测试"));                 //显示文本信息
05      CWnd* hParent = this->GetParent();
06      CenterWindow(hParent);                             //窗口居中
07      CRect rect;
08      this->GetClientRect(&rect);                        //获取客户区
09      int nEllipseWidth = rect.Width();                  //计算椭圆宽
10      int nEllipseHeight = rect.Height();                //计算椭圆高
11      //创建椭圆矩形
12      m rgnWnd.CreateEllipticRgn(0, 0, nEllipseWidth, nEllipseHeight);
13      SetWindowRgn((HRGN)m_rgnWnd, true);//将 m_rgnWnd 设置为对话框区域
14      return true;                                       //函数成功返回
15  }
```

上面代码会创建椭圆矩形的大小与对话框原有大小一致的椭圆形区域对话框，并将其显示在主程序界面的中间。程序运行效果如图 25-2 所示。

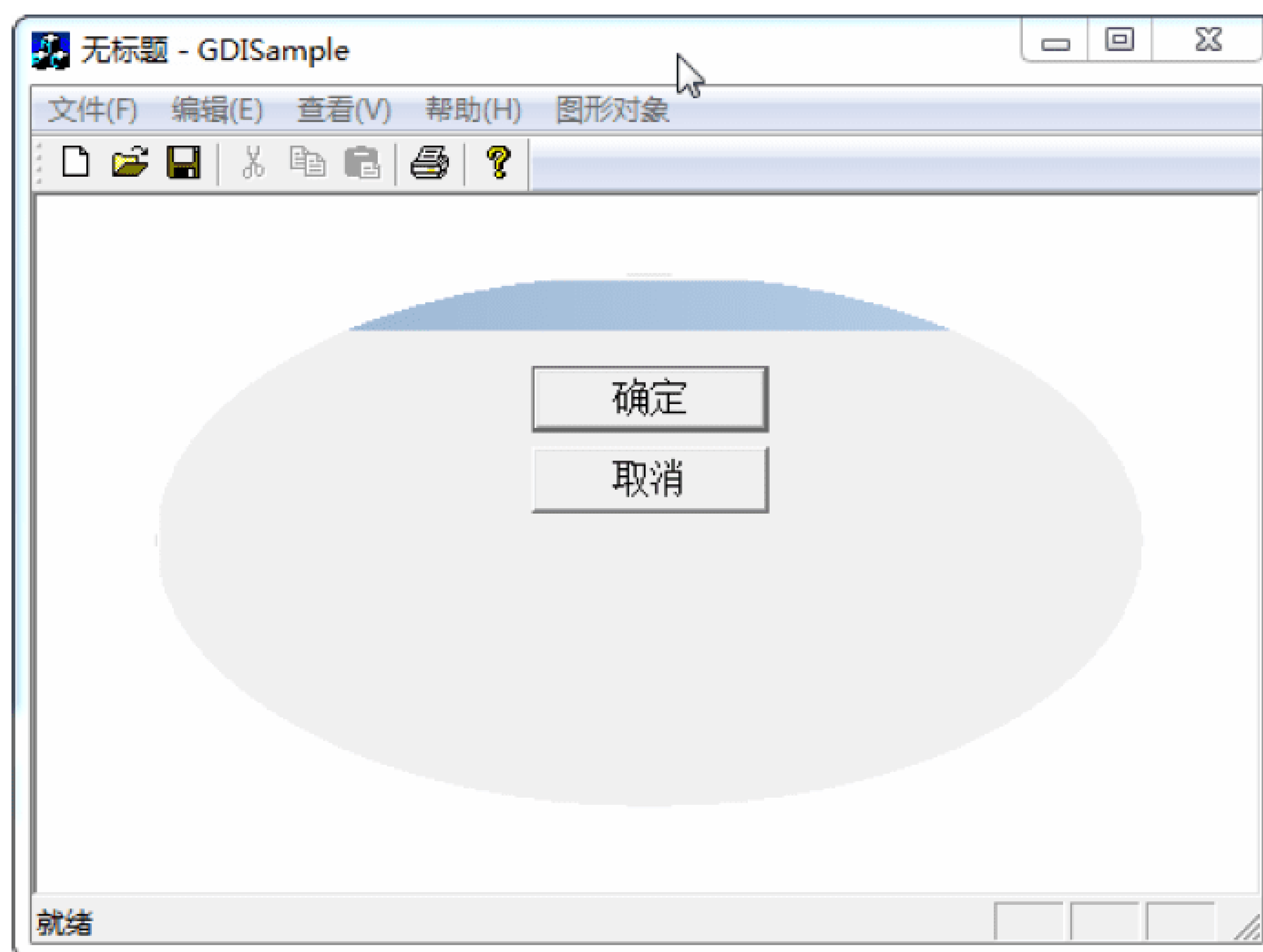


图 25-2　使用 CRgn 创建椭圆对话框的运行效果

25.2　画笔和画刷

第 24 章介绍了 Windows 中提供的 GDI 开发包处理有关图形的操作。其中，画笔用于描绘图形的轮廓，可以根据需要定制画笔的颜色和样式。画刷用于填充图形中的内容，可以根据需要填充画刷的样式。本节将介绍有关画笔和画刷的使用。

25.2.1　使用画笔对象

画笔对象的句柄类型为 HPEN，MFC 使用 CPen 封装了画笔对象，其包含一个 HPEN 类型的画笔句柄。使用其构造函数，可以创建指定样式的画笔对象。其函数原型为：

```
CPen(
  int nPenStyle,                 //指定画笔样式
  int nWidth,                    //指定画笔的宽度
  COLORREF crColor )             //指定画笔样式
CPen(
   int nPenStyle,                //指定画笔样式
  int nWidth,                    //指定画笔的宽度
  const LOGBRUSH* pLogBrush,     //指定 LOGBRUSH 结构
  int nStyleCount = 0,           //指定双字节单位的 lpStyle 数组长度
  const DWORD* lpStyle = NULL )  //指向双字节值的指针
```

其中，nPenStyle 参数指定画笔样式。其有效取值如下。

- PS_SOLID：创建实线画笔。
- PS_DASH：创建虚线画笔，此值只有当画笔的宽度等于或小于 1 时才有效。
- PS_DOT：创建点线画笔，此值只有当画笔的宽度等于或小于 1 时才有效。
- PS_DASHDOT：创建虚点线画笔，此值只有当画笔的宽度等于或小于 1 时才有效。
- PS_DASHDOTDOT：创建双点虚线画笔，此值只有当画笔的宽度等于或小于 1 时才有效。
- PS_NULL：创建空画笔。
- PS_INSIDEFRAME：创建封闭形状内线的画笔。
- PS_GEOMETRIC：创建几何画笔。
- PS_COSMETIC：创建装饰画笔。
- PS_ALTERNATE：创建设置每个像素的画笔，此类型只能应用于装饰画笔。
- PS_USERSTYLE：创建用户自定义样式的画笔。

线点的结束部分可以是下列取值。

- PS_ENDCAP_ROUND：结束部分是圆形。
- PS_ENDCAP_SQUARE：结束部分是方形。
- PS_ENDCAP_FLA：结束部分是平滑的。

连接点可以是下列取值。

- PS_JOIN_BEVEL：连接部分是斜角。
- PS_JOIN_MITER：连接部分在指定值内是连接的，否则是斜角。

- PS_JOIN_ROUND：连接部分是圆形。

如果使用没有参数的构造函数，则需要使用 CPen 类的 CreatePen()函数、CreatePenIndirect()函数或 CreateStockObject()成员函数初始化画笔。如果使用参数构造画笔，则不需要调用这些函数初始化画笔了。带参数的构造函数，如果发生错误时，会抛出异常；不带参数的初始化画笔不会抛出异常。

CreatePen()函数使用指定样式、宽度和画刷属性创建逻辑装饰画笔或几何画笔，并将其附加到 CPen 对象。CreatePenIndirect()函数在 LOGPEN 结构中指定样式、宽度和画刷属性创建逻辑装饰画笔或几何画笔，并将其附加到 CPen 对象。25.2.2 小节将通过实例介绍如何使用画笔绘图。

25.2.2　使用画笔绘图实例

本小节使用画笔，在屏幕对角线上绘制一条红色的实线。使用画笔绘图，首先需要创建用于进行图形绘制的画笔，可以指定画笔的样式、宽度和颜色。然后在 CDC 对象上使用 MoveTo()函数和 LineTo()函数绘制线的起点和终点。代码如下：

```
01  void CGDISampleView::OnMenuitemPen()                    //画笔实例
02  {
03      CDC* pDC = GetDC();                                 //获取设备上下文
04      CPen newPen;                                        //定义画笔对象
05      if( newPen.CreatePen( PS_SOLID, 2, RGB(255,0,0) ) )//创建画笔
06      {
07          CPen* pOldPen = pDC->SelectObject( &newPen );//装载新画笔
08          RECT rect;                                      //获取工作区
09          GetClientRect(&rect);
10          pDC->MoveTo(rect.left, rect.top);               //移动到起点
11          pDC->LineTo(rect.right, rect.bottom);           //绘制直线
12          pDC->SelectObject( pOldPen );                   //恢复画笔
13      }
14      else
15          MessageBox("创建画笔失败！");                   //输出错误信息
16  }
```

上面的代码使用 CreatePen()函数创建宽度为 2 的红色的实线，使用 GetDC()函数获取当前画布的句柄，并使用 CDC 对象的 MoveTo()函数和 LineTo()函数画出红线。其中使用 GetClientRect()函数获得画布的大小，并使用对角线的坐标画线。程序运行效果如图 25-3 所示。

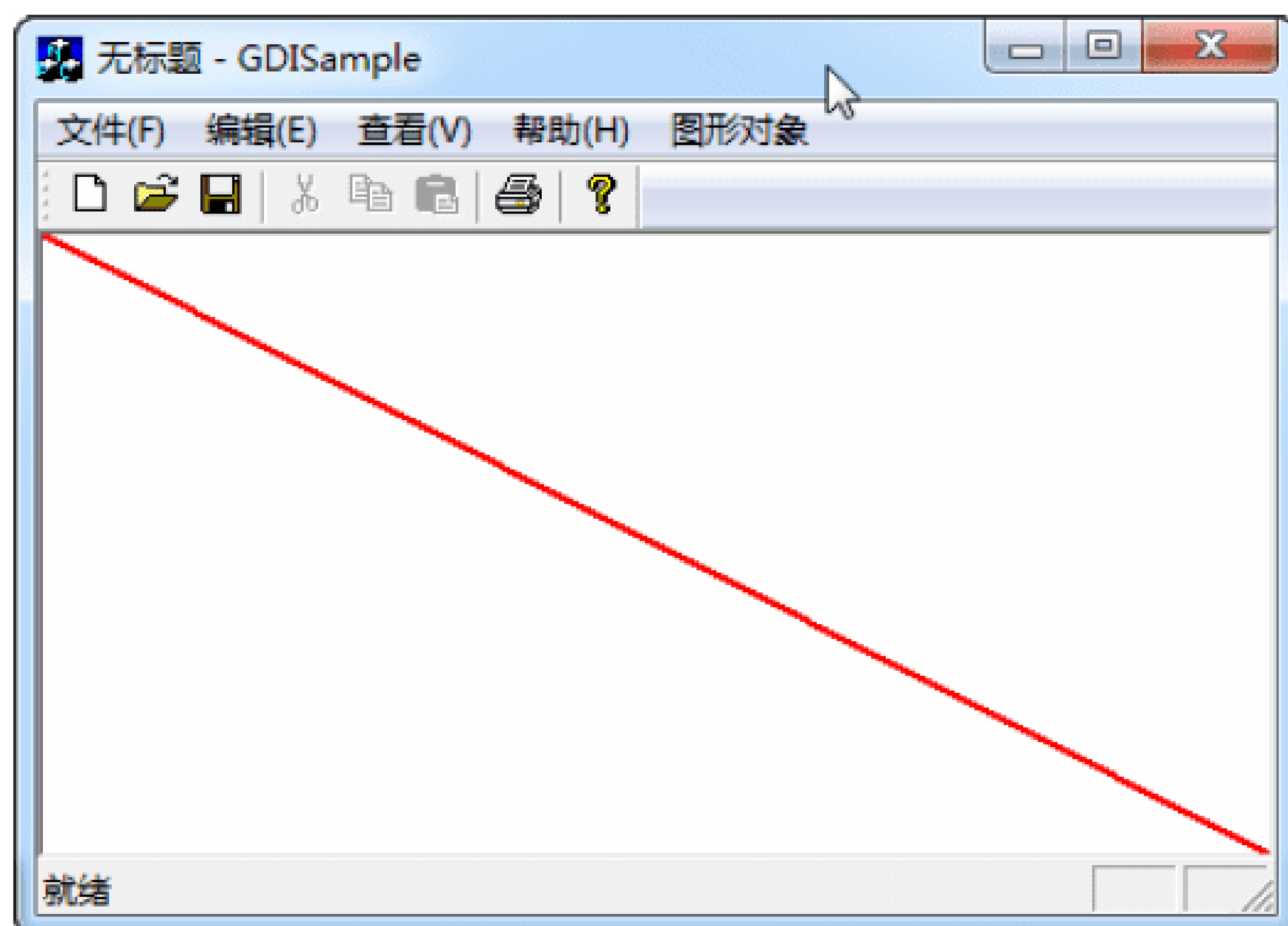


图 25-3　画笔绘图实例

25.2.3　使用画刷对象

MFC 在 CBrush 类中封装了画刷 Windows 图形设备接口，使用指定样式填充封闭区域。使用画刷对象构造 CBrush 对象，并将其传入任何 CDC 对象需要画刷的成员函数中。画刷可以是实画刷、纹理画刷和图案画刷。CBrush 类包含一个 HBRUSH 类型的画刷句柄。使用构造函数，可以创建指定样式的画刷对象。其函数原型为：

```
CBrush( COLORREF crColor )          //指定画刷的 RGB 前景颜色
CBrush( int nIndex,                 //指定画刷填充的样式
        COLORREF crColor )
CBrush( CBitmap* pBitmap )          //指定画刷用于绘制的位图对象的指针
```

其中，nIndex 参数指定画刷填充的样式，可以是以下几个取值。

- HS_BDIAGONAL：以左下角到右上角 45° 的线填充画刷。
- HS_CROSS：以十字交叉线填充画刷。
- HS_DIAGCROSS：以互相交互 45° 线填充。
- HS_FDIAGONAL：以左上角到右下角 45° 的线填充画刷。
- HS_HORIZONTAL：以水平线填充画刷。
- HS_VERTICAL：以垂直线填充画刷。

CBrush 类有 4 个构造函数。没有参数的构造函数不进行初始化，在使用前必须使用创建函数对画刷进行初始化。CBrush 提供的创建函数包括以下 6 个。

- CreateSolidBrush()函数：用于创建实画刷，即以指定的纯色填充区域。
- CreateHatchBrush()函数：用于创建纹理画刷，可以创建指定颜色的纹理画刷。
- CreateBrushIndirect()函数：使用在 LOGBRUSH 结构中指定样式、颜色和模式初始化画刷。
- CreatePatternBrush()函数：创建带有位图的画刷。
- CreateDIBPatternBrush()函数：使用设备相关位图创建画刷。
- CreateSysColorBrush()函数：创建系统默认颜色的画刷。

读者可以根据需要使用不同的构造函数或初始化函数创建画刷。25.2.4 小节将介绍如何使用画刷绘图。

25.2.4　使用画刷绘图实例

本小节使用画刷在屏幕中间部分绘制一个外接矩形比屏幕宽度小 20 像素的椭圆形。使用画刷绘图，首先需要创建用于进行图形绘制的画刷，可以指定画刷的样式和颜色。然后创建椭圆区域，并在 CDC 对象上填充椭圆区域。代码如下：

```
01  void CGDISampleView::OnMenuitemBrush()                   //画刷绘图实例
02  {
03      CDC* pDC = GetDC();                                  //获取设备上下文
04      CBrush newBrush;                                     //定义画刷对象
05      if( newBrush.CreateSolidBrush(RGB(0,255,0)))         //创建画刷
```

```
06      {
07          RECT rect;
08          GetClientRect(&rect);                       //获取工作区
09          CRgn rgn;                                   //定义矩形区域对象
10          rgn.CreateEllipticRgn(rect.left+10,rect.top,
11              rect.right-10,rect.bottom);             //创建矩形区域
12          pDC->FillRgn(&rgn, &newBrush);              //填充矩形
13      }
14      else
15          MessageBox("创建画刷失败！");                //输出错误信息
16  }
```

上面的代码使用 CreateSolidBrush()函数创建绿色的画刷，使用 GetDC()函数获取当前画布的句柄，并使用 CDC 对象的 FillRgn()函数画出绿色椭圆形。其中，使用 GetClientRect()函数获得画布的大小，并指定椭圆形的外接矩形的宽度比矩形小 20 个像素。程序运行效果如图 25-4 所示。

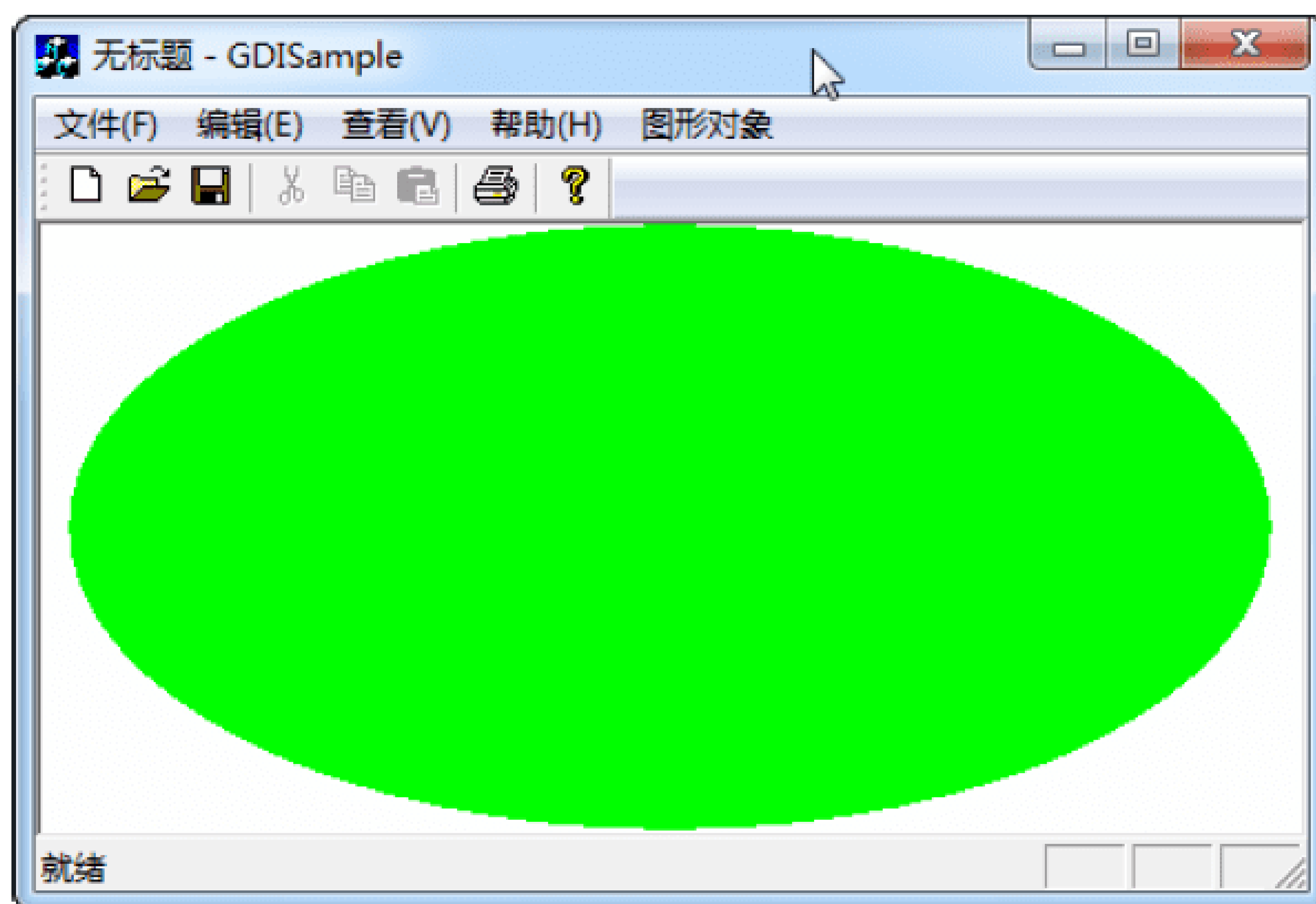


图 25-4　画刷绘图实例

25.3　图像基础技术

要进行图像方面的编程，需要对图像的核心原理和技术有深入的了解。本节本着学以致用的原则，以一些基本的图像处理为实例，逐步讲解图像基础技术，为后续的更深入层次的图像处理打下基础。因为 GDI+在 GDI 基础上提供了更方便的图像处理，同时，为了学习的渐进性，在本节中会穿插 GDI 和 GDI+两种编程方法。

25.3.1　如何使用 GDI+

25.2 节介绍的画笔和画刷是 Windows 图形设备接口 GDI 的内容，随着图形应用的复杂性，Windows 在原来的 GDI 基础上提供了 GDI+技术，它为 C/C++开发人员提供了一组基于类的用于处理图形和格式化文本的应用编程接口。GDI 可以应用于所有的 Windows 操作系统，GDI+在 Windows XP 和 Windows 2003 中是集成的，在其他的 Windows 平台下，需要安装 GDI+安装包，并且 GDI+支持 Win64 平台。

GDI+提供了 40 个类、50 个枚举、6 个结构体和几个公共函数。Graphics 是 GDI+的核心类，其他类会与此类结合使用，它是绘制线条、曲线、图形和图像的类。

要使用 GDI+，需要配置环境，步骤如下。

（1）如果使用 GDI+的应用程序所在的操作系统没有安装 GDI+，则需要将 gdiplus.dll 和 gdiplus.lib 文件复制到工程目录下，将 GDI+包含的头文件复制到 Visual Studio 2010 的头文件路径中。

（2）在需要使用 GDI+的文件中加入以下代码，包含 GDI+的头文件，并引入 Gdiplus 命名空间。

```
#define ULONG_PTR ULONG
#include <gdiplus.h>
using namespace Gdiplus;
#pragma comment( lib, "gdiplus.lib" )
```

（3）在应用程序类中定义 m_gdiplus 变量，代码如下：

```
ULONG_PTR m_gdiplus;
```

在应用程序类的 InitInstance()函数中加入以下代码，启动 GDI+工作：

```
Gdiplus::GdiplusStartupInput gdiplusStartupInput;
Gdiplus::GdiplusStartup(&m_gdiplus, &gdiplusStartupInput, NULL);
```

在应用程序类的 ExitInstance()函数中加入以下代码，清理 GDI+工作：

```
Gdiplus::GdiplusShutdown(m_gdiplus);
```

（4）在程序中加入要执行的 GDI+操作的代码，编译、调试、运行即可。

（5）在使用 GDI+程序时，要注意返回值的检测，代码如下：

```
01  Status status = GenericError;
02  Graphics graphics(m_hWnd);
03  status = graphics.GetLastStatus();
04  if (status != Ok)
05      return;
```

上面代码定义了 Status 类型的状态值，用于存储 GDI+操作的结果值。如果是 OK，则表示 GDI+操作成功，否则 GDI+操作失败，可以使用对象的 GetLastStatus()函数获取其值。读者在实际编程的过程中，应该判断每次 GDI+操作的返回值。在本书中，为了简化代码，去掉操作结果返回值的判断。

25.3.2　如何创建含有位图的画刷

前面介绍过画刷的使用，其中有种画刷是使用位图的画刷。使用此种画刷，填充封闭区域后，会在封闭区域的有效区域内显示位图的图像。位图画刷需要使用 CBitmap 对象构造。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemBmpbrush()  //创建含有位图的画刷示例
02  {
03      CDC* pDC = GetDC();
04      CRect rect;
05      GetClientRect(&rect);                     //获取工作区
```

```
06      rect.top = rect.Height()/4;                        //计算工作区
07      rect.bottom = rect.top*3;
08      rect.left = rect.Width()/4;
09      rect.right = rect.left*3;
10      CBitmap bitmap;                                    //定义位图变量
11      if (!bitmap.LoadBitmap(IDB_BITMAP_BABY))
12          return;                                        //装载位图
13      CBrush newBrush;                                   //定义画刷对象
14      if (newBrush.CreatePatternBrush(&bitmap))          //创建画刷
15          pDC->FillRect(&rect, &newBrush);               //使用位图画刷填充矩形
16  }
```

上面代码使用 GetClientRect()函数获取当前工作区的区域，然后定义了类型为 CBitmap 的位图变量，使用 CBitmap 类的 LoadBitmap()成员函数装载位图资源。最后定义了 CBrush 类型的画刷变量。使用画刷对象的 CreatePatternBrush()函数创建了位图画刷，并调用 CDC 设备上下文对象的 FillRect()函数填充定义的矩形区域。程序运行效果如图 25-5 所示。

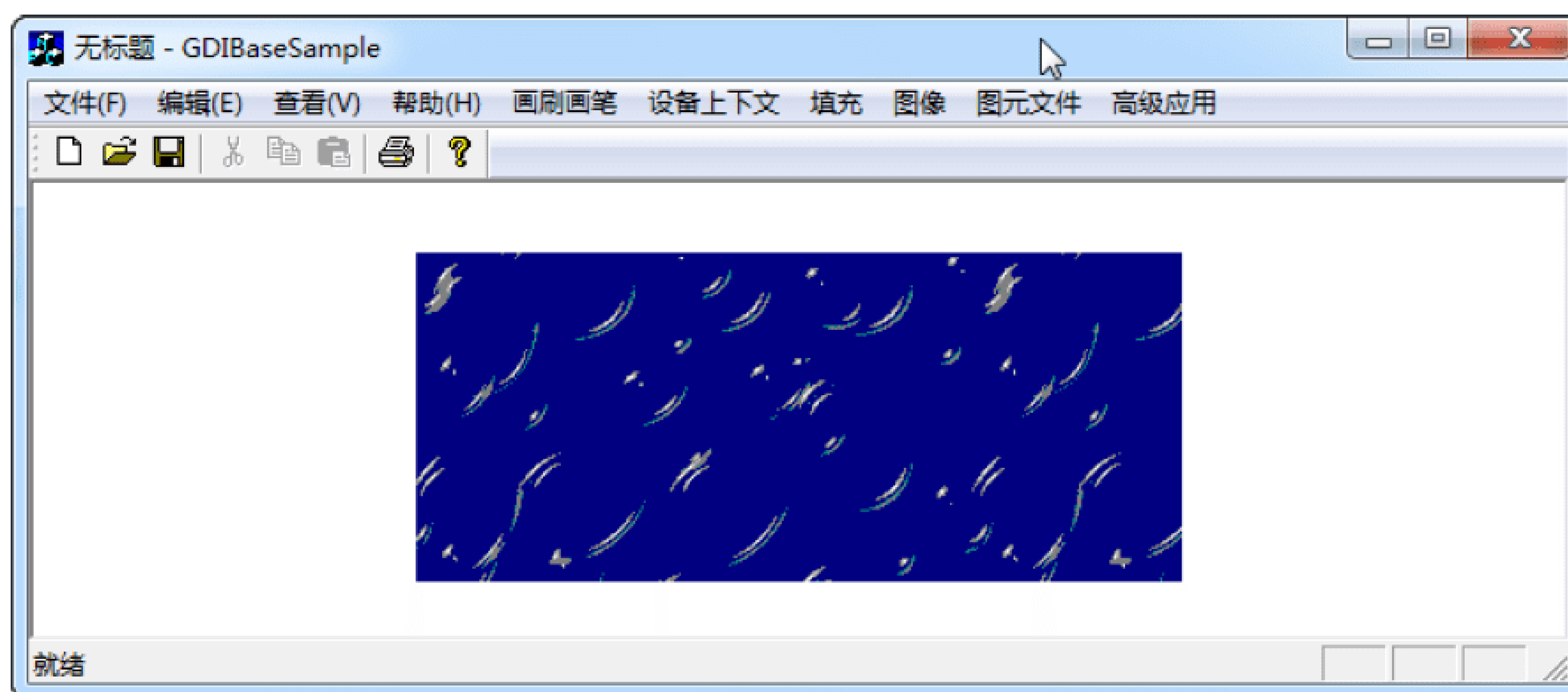


图 25-5　创建含有位图的画刷运行效果

25.3.3　保存屏幕抓图文件

通过 GDI+可以保存屏幕抓图文件。过程分为 3 步。第一步使用 CreateDC()函数获取屏幕抓图，并通过 GetSystemMetrics()函数计算屏幕的大小。第二步，将屏幕内容复制到 Bitmap 对象中。第三步，将 Bitmap 内容保存到文件中，并在屏幕上显示操作结果。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemSavescreentofile()
02  {
03      int cx = GetSystemMetrics(SM_CXSCREEN);        //获取屏幕分辨率
04      int cy = GetSystemMetrics(SM_CYSCREEN);
05      //创建显示屏上下文
06      HDC hScrDC = CreateDC("DISPLAY",  NULL,  NULL,  NULL);
07      Graphics graphics1(hScrDC);                     //定义 Graphics 对象
08      Bitmap  bitmap(cx, cy, &graphics1);             //定义位图对象
09      Graphics graphics2(&bitmap);                    //定义 Graphics 对象
10      HDC dc1 = graphics1.GetHDC();                   //获取设备上下文句柄
11      HDC dc2 = graphics2.GetHDC();                   //获取设备上下文句柄
12      BitBlt(dc2,0,0,cx,cy,dc1,0,0,13369376);         //获取位图
```

```
13      graphics1.ReleaseHDC(dc1);                    //释放设备上下文句柄
14      graphics2.ReleaseHDC(dc2);                    //释放设备上下文句柄
15      CLSID clsid;
16      char propertyValue[] = "屏幕截图";            //定义名称变量
17      PropertyItem* propertyItem = new PropertyItem;//创建属性项
18      //将抓到的图像保存为 jpg 文件
19      GetEncoderClsid(L"image/jpeg", &clsid);
20      propertyItem->id = PropertyTagImageTitle;
21      propertyItem->length = 16;
22      propertyItem->type = PropertyTagTypeASCII;
23      propertyItem->value = propertyValue;
24      bitmap.SetPropertyItem(propertyItem);
25      bitmap.Save(L"screen.jpg", &clsid, NULL);
26      CDC* pDC = GetDC();
27      pDC->TextOut("保存屏幕抓图到 screen.jpg 文件");
28  }
```

上面代码显示了保存屏幕抓图文件的过程，其中要注意 BitBlt()函数的使用和 Bitmap 类中 Save()函数的使用。程序运行效果如图 25-6 所示。



图 25-6　保存屏幕抓图文件运行效果

25.3.4　利用内存画布防止绘图时出现屏幕闪烁

在需要频繁向屏幕绘制图形的情况下，如果直接向屏幕绘制时间间隔较短，则会出现屏幕闪烁的情况。使用内存画布，即双缓冲可以消除闪烁的情况。因为直接向屏幕绘制，则每次向屏幕绘制新内容，屏幕都会重绘一次。如果两次重绘之间的时间间隔很短，则会频繁重绘，出现闪烁。而使用内存画布，则会先将要绘制的内容绘制到内存中，此块内存称为内存画布，等绘制完所有内容后，将内存画布中的内容一起绘制到屏幕上，因此就不会出现闪烁的情况。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemMemdc()        //内存画布示例
02  {
03      Bitmap bmp(300, 300);                         //定义位图对象
04      Graphics g(&bmp);                             //定义 Graphics 对象
05      Rect rect(0, 0, 300, 300);                    //矩形区域
06      //创建画刷对象
07      LinearGradientBrush brush(rect, Color.Green, Color.Blue,
08          LinearGradientModeHorizontal);
09      //循环输出矩形
10      for(int j = 0; j < 60; j ++)
11      {
12          for(int i = 0; i < 60; i++)
13          {
14              g.FillEllipse(&brush, i*5, j*5, 5, 5);
15          }
16      }
17      Graphics graphics(m_hWnd);                    //定义 Graphics
18      graphics.DrawImage(&bmp, 0, 0);               //绘制位图
19  }
```

上面的代码在内存画布中绘制渐变颜色，绘制完毕后，将内存画布的内容 bmp 一起绘制到屏幕上。程序运行效果如图 25-7 所示。

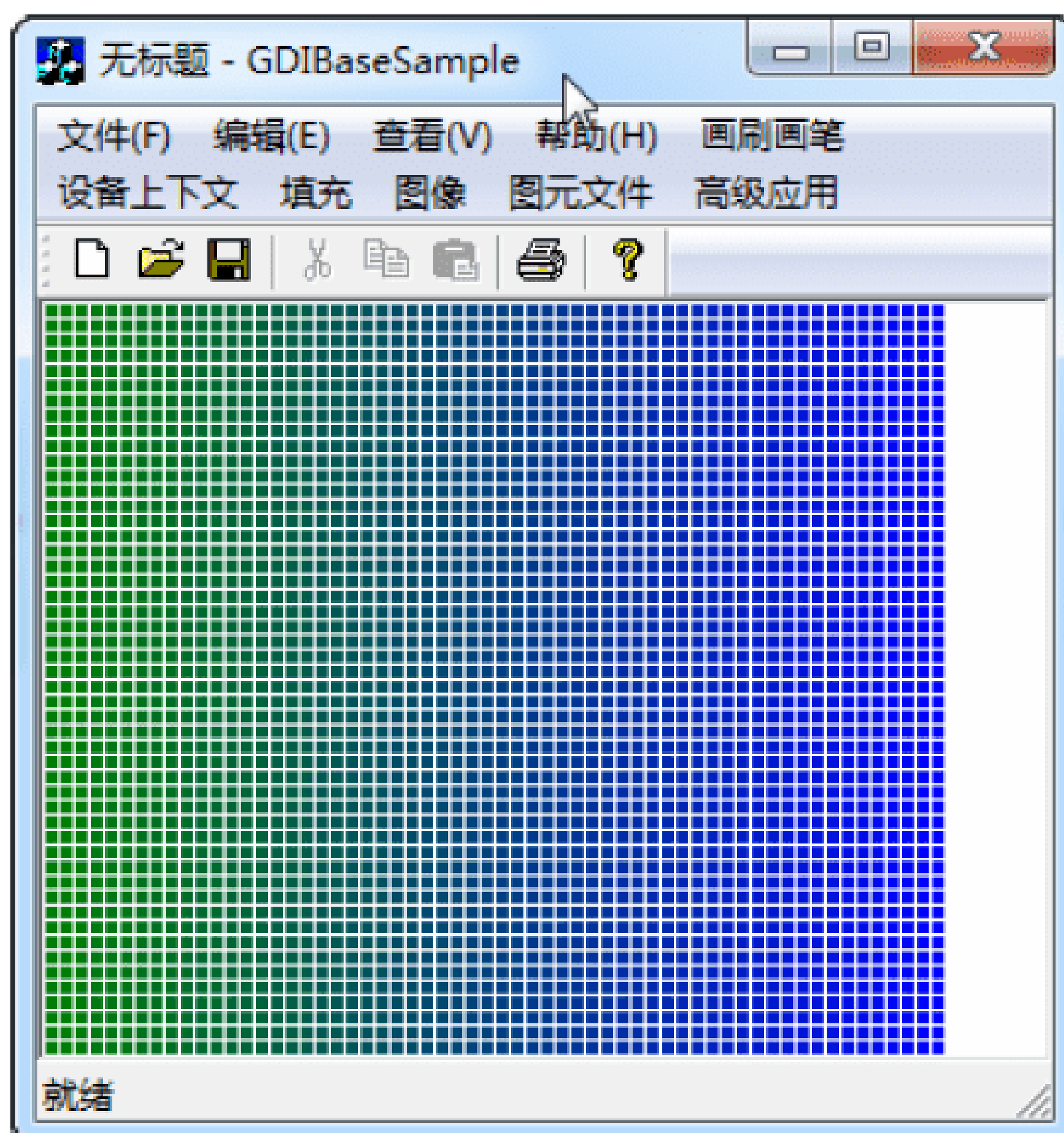


图 25-7　内存画布运行效果

25.3.5　创建几何画笔

使用几何画笔可以创建带有修饰的图形，如可以创建连接点是圆形的线段。本小节介绍如何使用 GDI 的 API 函数创建几何画笔，并将其附加到 CPen 对象中。使用 CPen 对象的函数是相同的，之所以使用 API 创建，是希望读者根据此实例中的代码，举一反三，在其他情况下，也可以将 MFC 的 GDI 类的成员函数与 GDI API 函数对应起来。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemGeometricpen()//几何画笔示例
02  {
03      CDC* pDC= GetDC();                                  //获取设备上下文
04      LOGBRUSH lb;                                        //定义几何画笔
05      lb.lbStyle = BS_SOLID;
06      lb.lbColor = RGB(0,0,255);
07      lb.lbHatch = HS_CROSS;
08      //创建几何画笔
09      HPEN hPen =
10          ExtCreatePen(PS_GEOMETRIC->PS_ENDCAP_SQUARE->PS_JOIN_ROUND,
11          10, &lb, 0,NULL);
12      if (hPen == NULL)
13          return;                                         //判断创建结果
14      CPen newPen;                                        //定义画笔
15      if(newPen.Attach(hPen))                             //附加画笔句柄
16      {
17          CPen* pOldPen = pDC->SelectObject( &newPen );//装载画笔
18          CRect rect;
19          GetClientRect(&rect);
20          rect.top = rect.Height()/4;
21          rect.bottom = rect.top*3;
22          rect.left = rect.Width()/4;
23          rect.right = rect.left*3;
24          pDC->Rectangle(&rect);                          //使用新画笔绘制矩形
25      }
26  }
```

上面的代码定义了 LOGBRUSH 结构的变量，并指定线条的样式、颜色和纹理类型，此处指定创建蓝色的、实线的十字交叉填充的画笔。定义了 HPEN 类型的 hPen 变量，并使用 ExtCreatePen()函数创建几何画笔，PS_GEOMETRIC 选项表示创建几何画笔，PS_ENDCAP_SQUARE 表示结束点是方形，PS_JOIN_ROUND 选项表示中间连接点是圆角的。创建成功后，将其附加到 CPen 对象中，并将其选择到 CDC 对象中。然后通过 GetClientRect()函数获取工作区范围，并使用 CDC 类的 Rectangle 成员函数在画布上绘制矩形区域。程序运行效果如图 25-8 所示。

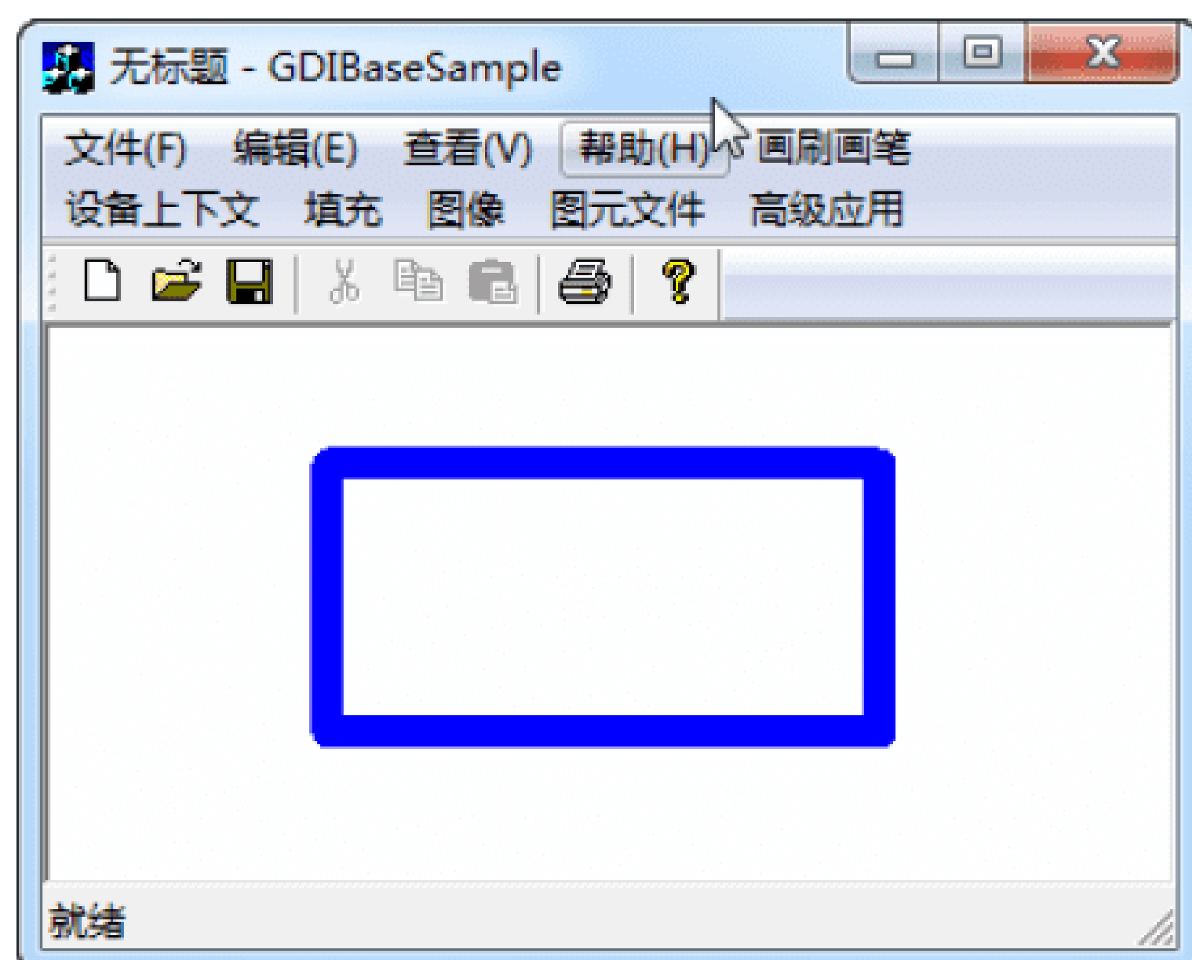


图 25-8　创建几何画笔运行效果

25.3.6　绘制网格

通过画笔绘制多条交叉直线的方式，可以实现绘制网格，网格在棋盘的绘制、图形坐标的绘制方面都非常有用。本小节以一个实例，讲解如何在画布上绘制指定横格格数和纵格格数的网格。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemDrawCrossline()//绘制网格示例
02  {
03      CDC* pDC= GetDC();                                  //获取设备上下文
04      CPen newPen;                                        //创建画笔
05      if( newPen.CreatePen( PS_SOLID, 2, RGB(125,125,125) ) )
06      {
07          const int HC = 9;
08          const int VC = 9;
09          CPen* pOldPen = pDC->SelectObject( &newPen );//装载画笔
10          CRect rect;
11          GetClientRect(&rect);                           //获取工作区
12          int dx = rect.Width()/(HC-1);
13          int dy = rect.Height()/(VC-1);
14          CPoint (*Point)[VC] = new CPoint[HC][VC];
15          for(int i=0;i<HC;i++)                  //在工作区中依次绘制横线和竖线
16          {
17              for(int j=0;j<VC;j++)
18              {
19                  Point[i][j].x =  i*dx;
20                  Point[i][j].y =  j*dy;
21              }
22          }
```

```
23          for(i=0;i<HC;i++)
24          {
25              pDC->MoveTo(Point[i][0]);
26              pDC->LineTo(Point[i][VC-1]);
27          }
28          for(int j=0;j<VC;j++)
29          {
30              pDC->MoveTo(Point[0][j]);
31              pDC->LineTo(Point[HC-1][j]);
32          }
33          pDC->SelectObject( pOldPen );
34      }
35  }
```

上面代码创建了灰色的宽度为 2 像素的实线画笔。通过 GetClientRect()函数获取画布的范围，根据指定的横格格数和纵格格数将画布范围从横轴方向和纵轴方向平均分配，此实例中创建 9 条横线和 9 条纵线相交的网格。然后使用 CDC 对象的 LineTo()函数分别绘制横线和纵线。绘制完网格后，不要忘记恢复原来的画笔。程序运行效果如图 25-9 所示。

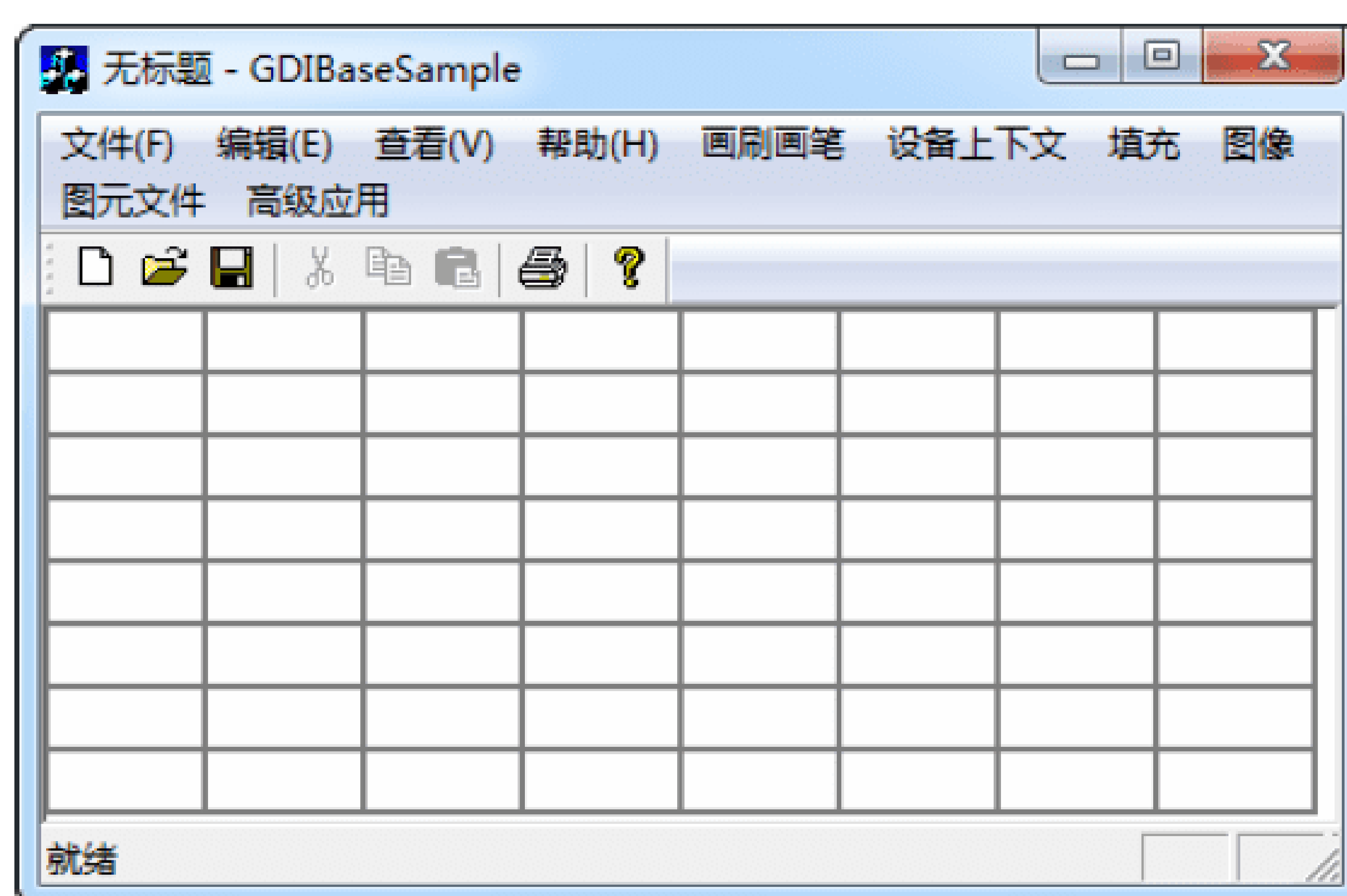


图 25-9　绘制网格运行效果

25.3.7　创建不同的画刷

前面介绍过画刷的使用，本小节介绍几种画刷的使用，包括纯色画刷、纹理画刷和系统默认颜色的画刷。具体代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemMultibrush()
02  {
03      CDC* pDC = GetDC();                               //获取设备上下文
04
05      CBrush newBrush;                                  //创建画刷
06      newBrush.CreateSolidBrush(RGB(255, 255, 0));
07      CRect rect;
08      GetClientRect(&rect);
09      int width = rect.Width()/4;
10      rect.right = width;
11      pDC->FillRect(&rect, &newBrush);                  //使用画刷填充矩形
12      ::DeleteObject((HGDIOBJ)newBrush);
13
14      CBrush newBrush1;                                 //创建纹理画刷
15      newBrush1.CreateHatchBrush(HS_CROSS, RGB(0,255,255));
```

```
16      rect.left += width;
17      rect.right += width;
18      pDC->FillRect(&rect, &newBrush1);          //使用纹理画刷填充矩形
19      ::DeleteObject((HGDIOBJ)newBrush1);
20      LOGBRUSH logBrush;
21      ogBrush.lbColor = RGB(255, 0, 255);
22      logBrush.lbHatch = HS_HORIZONTAL;
23      logBrush.lbStyle = BS_HATCHED;
24
25      CBrush newBrush2;
26      newBrush2.CreateBrushIndirect(&logBrush);  //创建纹理画刷
27      rect.left += width;
28      rect.right += width;
29      pDC->FillRect(&rect, &newBrush2);          //使用纹理画刷填充矩形
30      ::DeleteObject((HGDIOBJ)newBrush2);
31
32      CBrush newBrush3;
33      newBrush3.CreateSysColorBrush(HS_VERTICAL);//创建系统颜色的画刷
34      rect.left += width;
35      rect.right += width;
36      pDC->FillRect(&rect, &newBrush3);      //使用系统颜色的画刷填充矩形
37      ::DeleteObject((HGDIOBJ)newBrush3);
38  }
```

上面的代码中，newBrush 变量使用 CreateSolidBrush()函数创建黄色的纯色画刷；newBrush1 变量使用 CreateHatchBrush()函数创建蓝色的十字交叉的纹理画刷；newBrush2 变量使用 CreateBrushIndirect()函数创建粉红色的使用横线填充的纹理画刷；newBrush3 变量使用 CreateSysColorBrush()函数创建使用系统默认颜色的垂直填充的画刷。读者可以根据需要创建颜色不同、样式不同的各种类型的画刷。程序运行效果如图 25-10 所示。

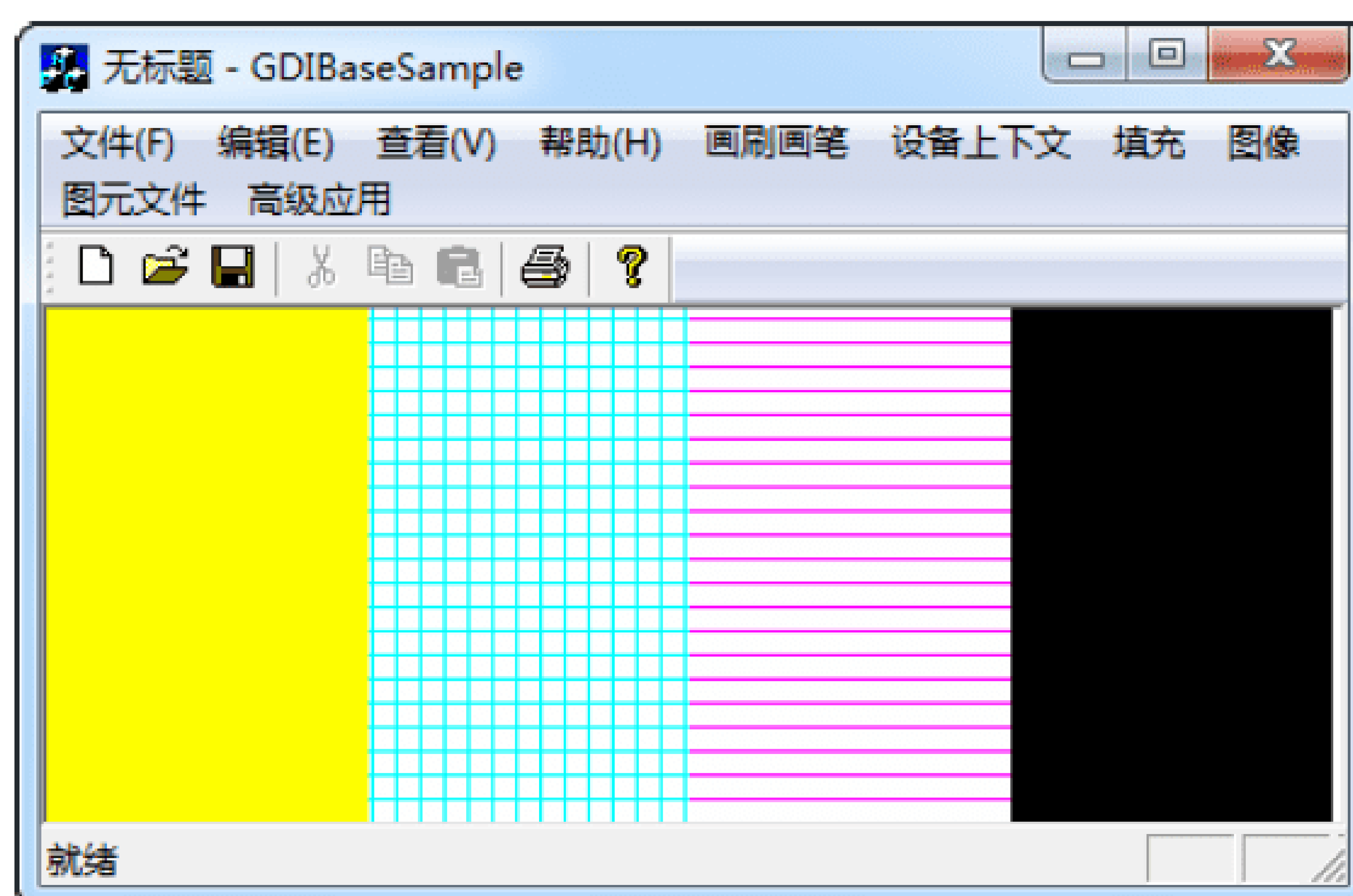


图 25-10　创建不同的画刷运行效果

25.3.8　填充矩形区域

使用画刷可以填充矩形区域，最简单的办法就是使用带有颜色参数的 CBrush 类的初始化构造函数。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemFillRect()
02  {
03      CDC* pDC= GetDC();                          //获取设备上下文
```

```
04      CBrush backBrush(RGB(255, 128, 128));       //创建画刷
05      CRect rect;
06      GetClientRect(&rect);
07      rect.top = rect.Height()/4;
08      rect.bottom = rect.top*3;
09      rect.left = rect.Width()/4;
10      rect.right = rect.left*3;
11      pDC->FillRect(&rect, &backBrush);           //使用画刷填充矩形
12  }
```

上面代码使用带参数的 CBrush 构造函数创建了纯色画刷对象。使用 GetClientRect()函数获取当前画布的工作区域，根据此区域，计算要填充的矩形区域的范围。本实例中在画布中间填充的是其范围一半的矩形区域。最后，使用 CDC 对象的 FillRect()函数使用创建的纯色画刷填充矩形区域。程序运行效果如图 25-11 所示。

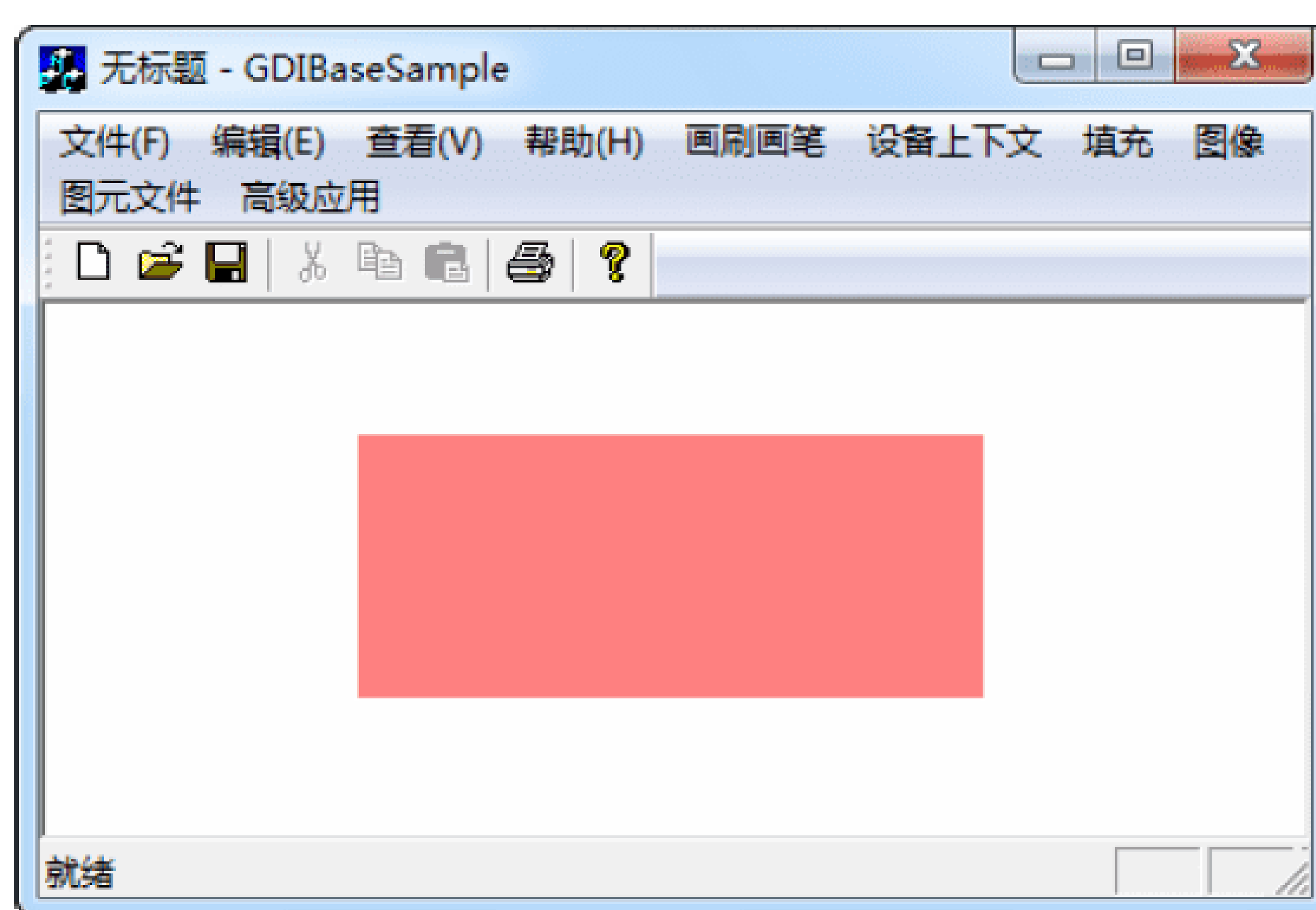


图 25-11　填充矩形区域运行效果

25.3.9　模拟时钟

根据当前时间和钟表的显示原理，可以使用 GDI+接口模拟时钟的实现。本小节中的时钟启动一个计时器，每隔一秒钟，根据当前时间重绘一次屏幕，在时钟上显示当前时间。代码如下：

```
01  void CGDIBaseSampleView::OnTimer(UINT nIDEvent)    //定时处理函数
02  {
03      if (nIDEvent == 200)                           //判断定时器编号
04      {                                              //绘制钟表盘
05          Graphics graphics(m hWnd);                 //定义 Graphics
06          int width = 300;
07          int height = 300;
08          Rect outRect(0, 0, width, height);         //定义输出矩形
09          Rect midRect(6, 6, 288, 288);              //定义中间矩形
10          Rect inRect(9, 9, 282, 282);               //定义输入矩形
11          LinearGradientBrush outBrush(outRect,
12              Color(0, 125, 0), Color(0, 255, 0),
13              LinearGradientModeBackwardDiagonal);   //创建输出区域画刷
14          LinearGradientBrush midBrush(midRect,
15              Color(0, 255, 0), Color(0, 125, 0),
16              LinearGradientModeBackwardDiagonal);   //创建中间区域画刷
17          LinearGradientBrush inBrush(inRect,
18              Color(0, 125, 0), Color(0, 255, 0),
19              LinearGradientModeBackwardDiagonal);   //创建输入区域画刷
```

```
20          graphics.FillEllipse(&outBrush, outRect);     //填充输出矩形
21          graphics.FillEllipse(&midBrush, midRect);     //填充中间矩形
22          graphics.FillEllipse(&inBrush, inRect);       //填充输入矩形
23          FontFamily fontFamily(L"Arial");              //绘制刻度
24          Font font(&fontFamily, 20, FontStyleBold, UnitPixel);
25          SolidBrush whiteBrush(Color(255,255,255,255));//创建画刷
26          graphics.DrawString(L"12", -1, &font,
27              PointF(130, 10), &whiteBrush);
28          graphics.DrawString(L"6", -1, &font,
29              PointF(140, 265), &whiteBrush);
30          graphics.DrawString(L"3", -1, &font,
31              PointF(270, 140), &whiteBrush);
32          graphics.DrawString(L"9", -1, &font,
33              PointF(10, 140), &whiteBrush);
34          graphics.DrawString(L"1", -1, &font,
35              PointF(200, 30), &whiteBrush);
36          graphics.DrawString(L"2", -1, &font,
37              PointF(250, 80), &whiteBrush);
38          graphics.DrawString(L"5", -1, &font,
39              PointF(205, 245), &whiteBrush);
40          graphics.DrawString(L"4", -1, &font,
41              PointF(250, 200), &whiteBrush);
42          graphics.DrawString(L"11", -1, &font,
43              PointF(65, 30), &whiteBrush);
44          graphics.DrawString(L"10", -1, &font,
45              PointF(20, 80), &whiteBrush);
46          graphics.DrawString(L"7", -1, &font,
47              PointF(65, 245), &whiteBrush);
48          graphics.DrawString(L"8", -1, &font,
49              PointF(25, 200), &whiteBrush);
50          graphics.TranslateTransform(150, 150,
51              MatrixOrderAppend);  //绘制指针
52          Pen hourPen(Color(255, 0, 255, 0), 7);            //时针
53          hourPen.SetLineCap(LineCapRoundAnchor,
54              LineCapArrowAnchor, DashCapFlat);
55          Pen minutePen(Color(255, 0, 0,255), 4);           //分针
56          minutePen.SetLineCap(LineCapRoundAnchor,
57              LineCapArrowAnchor, DashCap
58              Flat);
59          Pen secondPen(Color(255, 255, 0, 0), 2);          //秒针
60          CTime time = CTime::GetCurrentTime();             //获取当前时间
61          int sec = time.GetSecond();
62          int min = time.GetMinute();
63          int hour = time.GetHour();
64          //计算当前时间各个分量角度值
65          const double pi = 3.1415926;
66          double secondAngle = 2.0 * pi * sec / 60.0;
67          double minuteAngle = 2.0 * pi * (min + sec / 60.0) / 60.0;
68          double hourAngle = 2.0 * pi * (hour + min / 60.0) / 12.0;
69          Point centre(0, 0);
70          Point hourHand((int)(40 * sin(hourAngle)),
71              (int)(-40 * cos(hourAngle)));
72          graphics.DrawLine(&hourPen, centre, hourHand);  //绘制时针
73          Point minHand((int)(80 * sin(minuteAngle)),
74              (int)(-80 * cos(minuteAngle)));
75          graphics.DrawLine(&minutePen, centre, minHand); //绘制分针
76          Point secHand((int)(80 * sin(secondAngle)),
77              (int)(-80 * cos(secondAngle)));
78          graphics.DrawLine(&secondPen, centre, secHand); //绘制秒针
79      }
80      CView::OnTimer(nIDEvent);                     //调用定时器基类处理函数
81  }
```

上面代码显示了如何绘制底盘为绿色，刻度为白色，时针为绿色，分针为蓝色，秒针为红色的时钟。创建过程分为 3 个步骤。第一步是绘制表盘，在本例中，绘制 3 个大小略有不同的颜色渐变椭圆，实现带有边的时钟表盘。第二步是绘制刻度，根据计算得出的结果使用白色画笔在时间刻度对应的位置上绘制小时刻度。读者可以根据此例，自己扩展，在表盘上绘制分钟刻度。第三步是根据当前时间，绘制 3 个指针，此处用到数学公式来计算指针所在的位置。程序运行效果如图 25-12 所示。

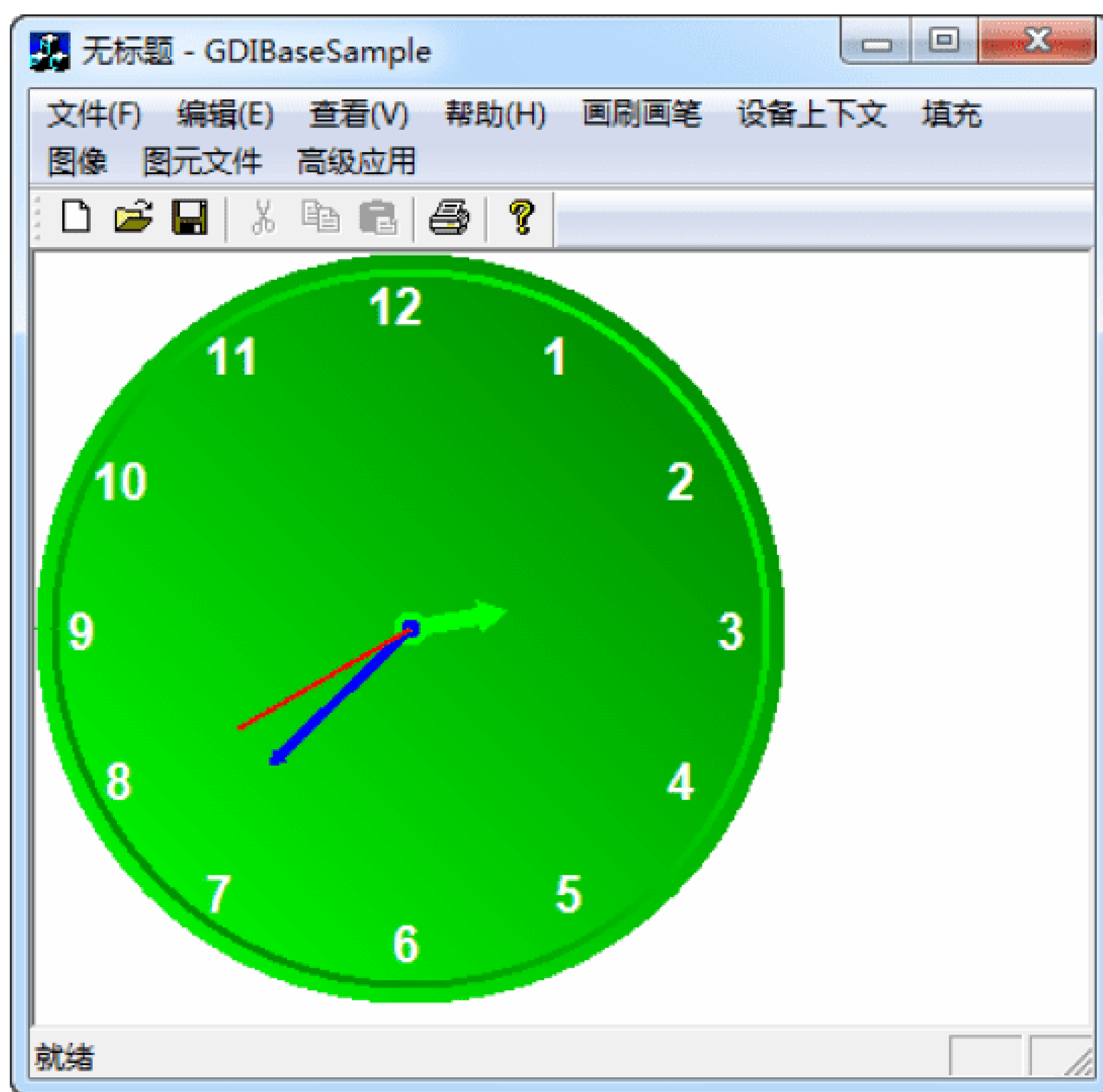


图 25-12　模拟时钟运行效果

25.3.10　颜色渐变算法

颜色渐变的算法有多种，究其原则就是颜色过渡要平滑规律，这样颜色渐变的效果才好。本小节以一个实例介绍颜色渐变的算法的实现，此实例中会在程序画布上以 255×255 像素矩阵的方式显示颜色点像素。代码如下。

```
01  void CGDIBaseSampleView::OnMenuitemColorchange()   //颜色渐变示例
02  {
03      CDC* pDC= GetDC();                              //获取设备上下文
04      int  i=0,j=0;
05      //使用 for 循环处理颜色渐变中的各个像素点的颜色值
06      for(i=0;i<255;i++)
07      {
08          for(j=0;j<255;j++)
09          {
10              DWORD dwColor = (unsigned long)(0x00000000|0|j<<8|i);
11              pDC->SetPixel(i,j,dwColor);
12          }
13      }
14  }
```

在上面代码中，颜色渐变算法是实现红色和绿色两种颜色的颜色渐变。RGB 颜色是红色（Red）、绿色（Green）和蓝色（Blue）3 种元色组合而成的颜色，也就是三元色。RGB 颜色值实际上是一个 DWORD 值 0x00000000。其中，第一个字节表示红色值，第二个字

节值表示绿色值，第三个字节值表示蓝色值，这 3 个值组合起来就是颜色值。本例中，使用当前像素点在矩阵中的横坐标值和纵坐标值分别定义了红色值和绿色值。在矩阵中指定像素点显示对应的颜色值。程序运行效果如图 25-13 所示。

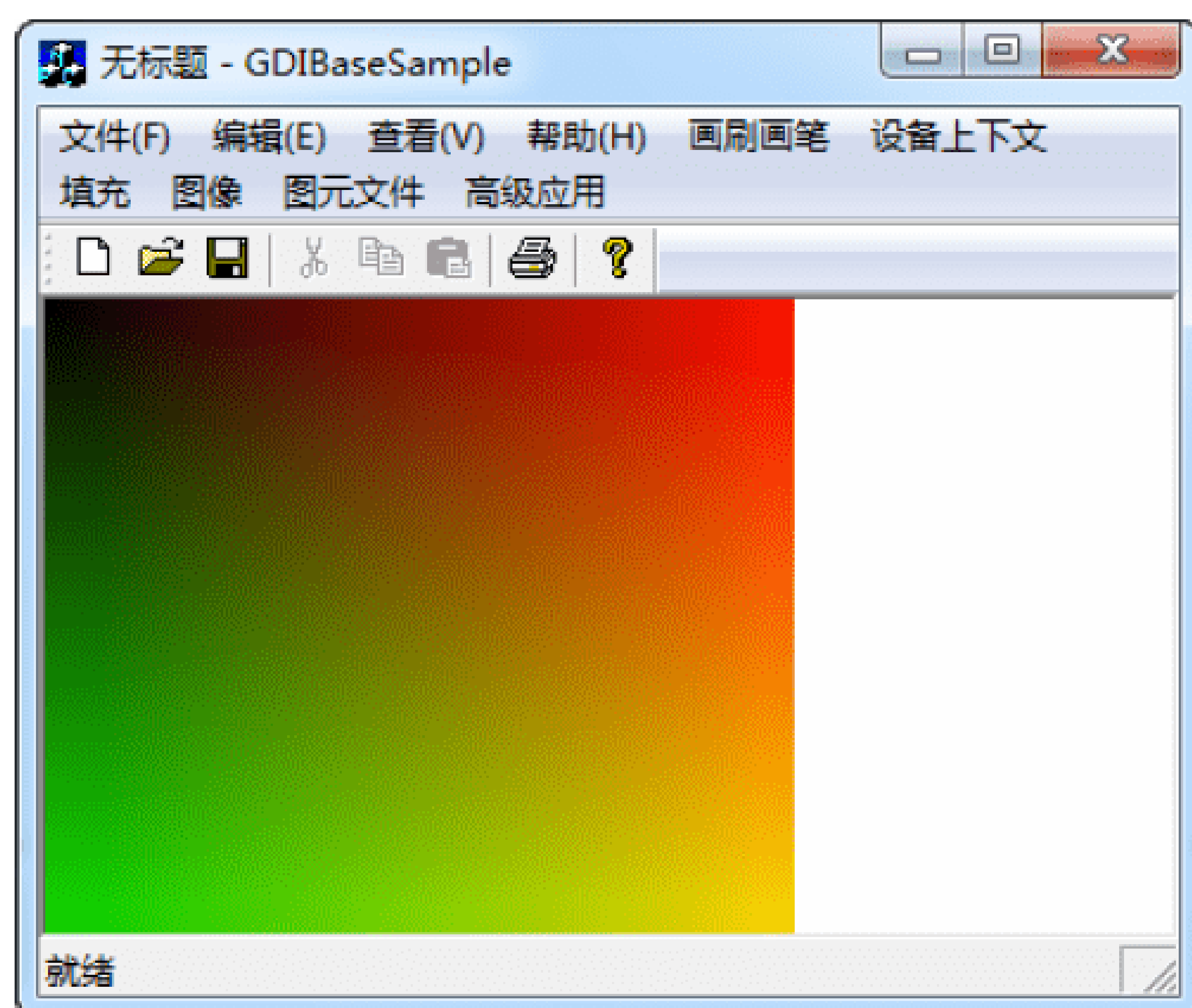


图 25-13　颜色渐变算法运行效果

25.3.11　如何绘制渐变颜色

使用 GDI API 函数 GradientFill()可以填充矩形和三角形结构，并实现绘制渐变颜色。其函数原型为：

```
BOOL GradientFill(
  HDC hdc,                    //目标设备上下文句柄
  CONST PTRIVERTEX pVertex, //指向 TRIVERTEX 结构的数组，用于定义三角矢量
  DWORD dwNumVertex,          //表示三角矢量点的个数
  CONST PVOID pMesh,          //指定点的信息
  DWORD dwNumMesh,            //pMesh 参数中的元素数目
  DWORD dwMode);     //指定填充模式，分为垂直填充矩形、水平填充矩形和填充三角形
```

如果函数成功，则返回 true；如果函数失败，则返回 false，通过 GetLastError()函数可以获得错误原因。下面代码显示了如何使用此函数绘制渐变颜色。

```
01  //绘制渐变颜色示例
02  void CGDIBaseSampleView::OnMenuitemDrawColorchange()
03  {
04      CDC* pDC=GetDC();             //获取设备上下文
05      CRect rect;
06      GetClientRect(&rect);         //获取工作区
07      TRIVERTEX vert[2];            //定义矢量变量
08      GRADIENT_RECT gRect;
09      vert[0].x = rect.left;
10      vert[0].y = rect.top;
11      vert[0].Red = 0xff00;
12      vert[0].Green = 0x0000;
13      vert[0].Blue = 0x0000;
14      vert[0].Alpha = 0;
15      vert[1].x = rect.right;
16      vert[1].y = rect.bottom;
```

```
17      vert[1].Red = 0x0000;
18      vert[1].Green = 0xff00;
19      vert[1].Blue = 0x0000;
20      vert[1].Alpha = 0;
21      gRect.UpperLeft=0;
22      gRect.LowerRight=1;
23      //绘制渐变矩形
24      GradientFill(pDC->GetSafeHdc(),vert,2,&gRect,
25                     1,GRADIENT_FILL_RECT_H);
26  }
```

上面代码通过定义 TRIVERTEX 类型的数组，实现左端从红色、右端从绿色的渐变色，并填充整个画布区域。程序运行效果，如图 25-14 所示。

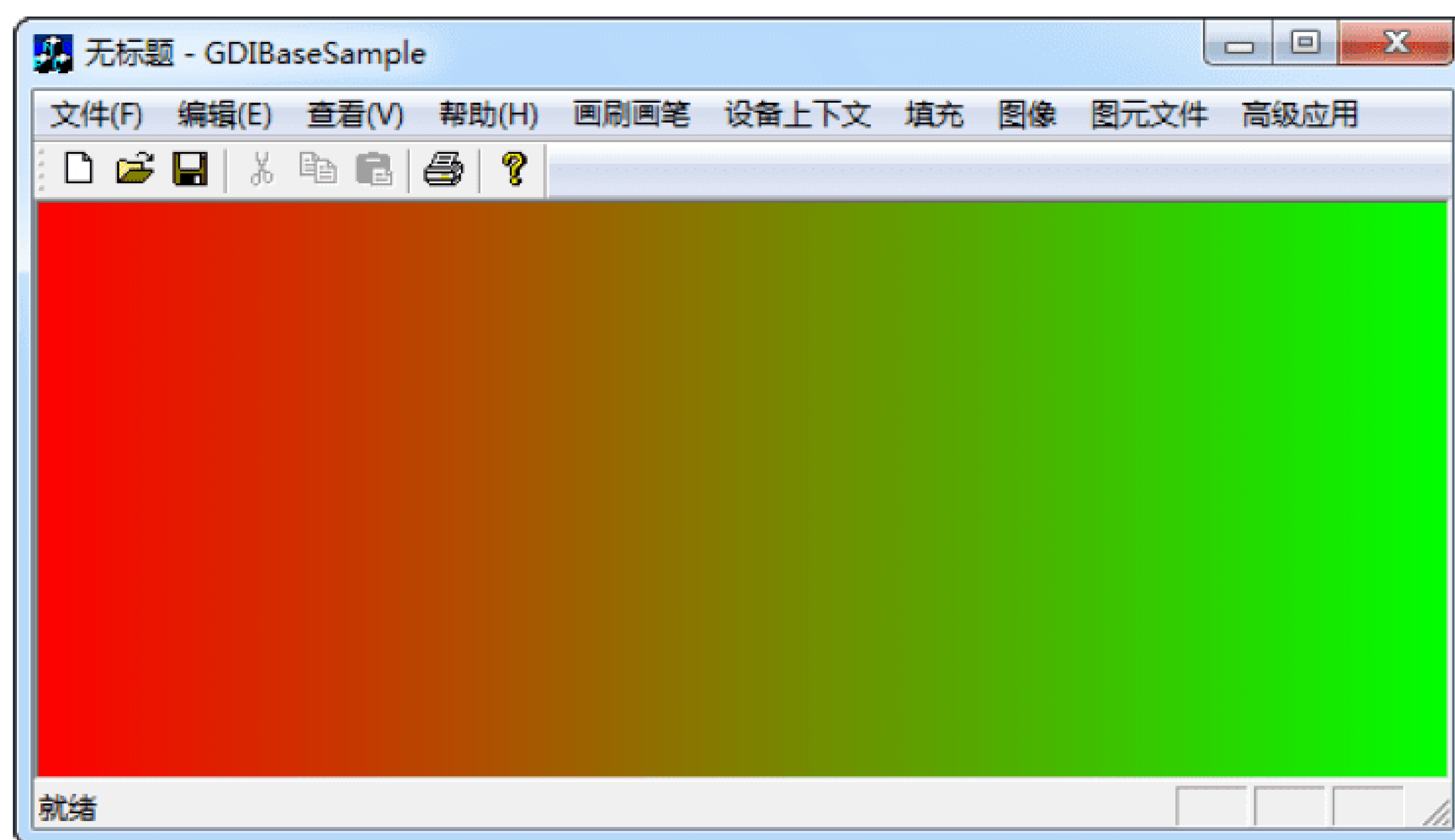


图 25-14　绘制渐变颜色运行效果

25.3.12　图元文件的保存与打开

图元文件即矢量文件，是存储设备无关格式图片的结构集合。设备无关是元文件与位图区分的一个主要属性。与位图不同，源文件保证设备无关性，可以将其写回源文件，但是绘制速度比位图要慢。GDI+中提供 Metafile 保存和显示图元文件，Graphics 对象需要与 Metafile 对象相连。此对象支持的图元文件的扩展名是 emf 和 emf+格式。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemSavemetafile()//图元文件示例
02  {
03      CDC* pDC = GetDC();                         //获取设备上下文
04      Status status = GenericError;
05      Metafile metafile(L"MFSample.emf", pDC->m_hDC);//定义图元文件变量
06      {
07          Graphics graphics(&metafile);           //装载 Graphics
08          status = graphics.GetLastStatus();      //获取装载状态
09          if (status != Ok)
10              return;
11          Pen pen(Color(255, 0, 255, 0));         //创建画笔
12          status = pen.GetLastStatus();           //获取创建状态
13          if (status != Ok)
14              return;
15          SolidBrush solidBrush(Color(255, 0, 255, 255));//创建画刷
```

```
16          status = solidBrush.GetLastStatus();     //获取创建状态
17          if (status != Ok)
18              return;
19          CRect rect;
20          GetClientRect(&rect);                    //获取工作区
21          int nLeft = (rect.Width())/2;
22          int nTop = (rect.Height())/2;
23          graphics.DrawRectangle(&pen, Rect(0, 0, nLeft, nTop));
24          graphics.FillEllipse(&solidBrush, Rect(nLeft,
25              nTop, nLeft/2,nTop/2));              //填充椭圆形
26          graphics.SetSmoothingMode(SmoothingModeHighQuality);
27      }
28      Graphics graphics(pDC->m_hDC);               //定义 Graphics
29      graphics.DrawImage(&metafile, 0, 0);         //绘制图像
30  }
```

上面代码定义了名称为 MFSample.emf 的图元文件对象 metafile，并将其与当前画布句柄关联起来，接下来进行图形的绘制，此时 metafile 文件中会保存接下来的图形绘制指令序列。绘制完成后。重新将 Graphics 与当前画布关联起来，并显示图元文件。也可以使用 Image 对象像显示其他格式的图像一样显示图元文件。程序运行效果如图 25-15 所示。

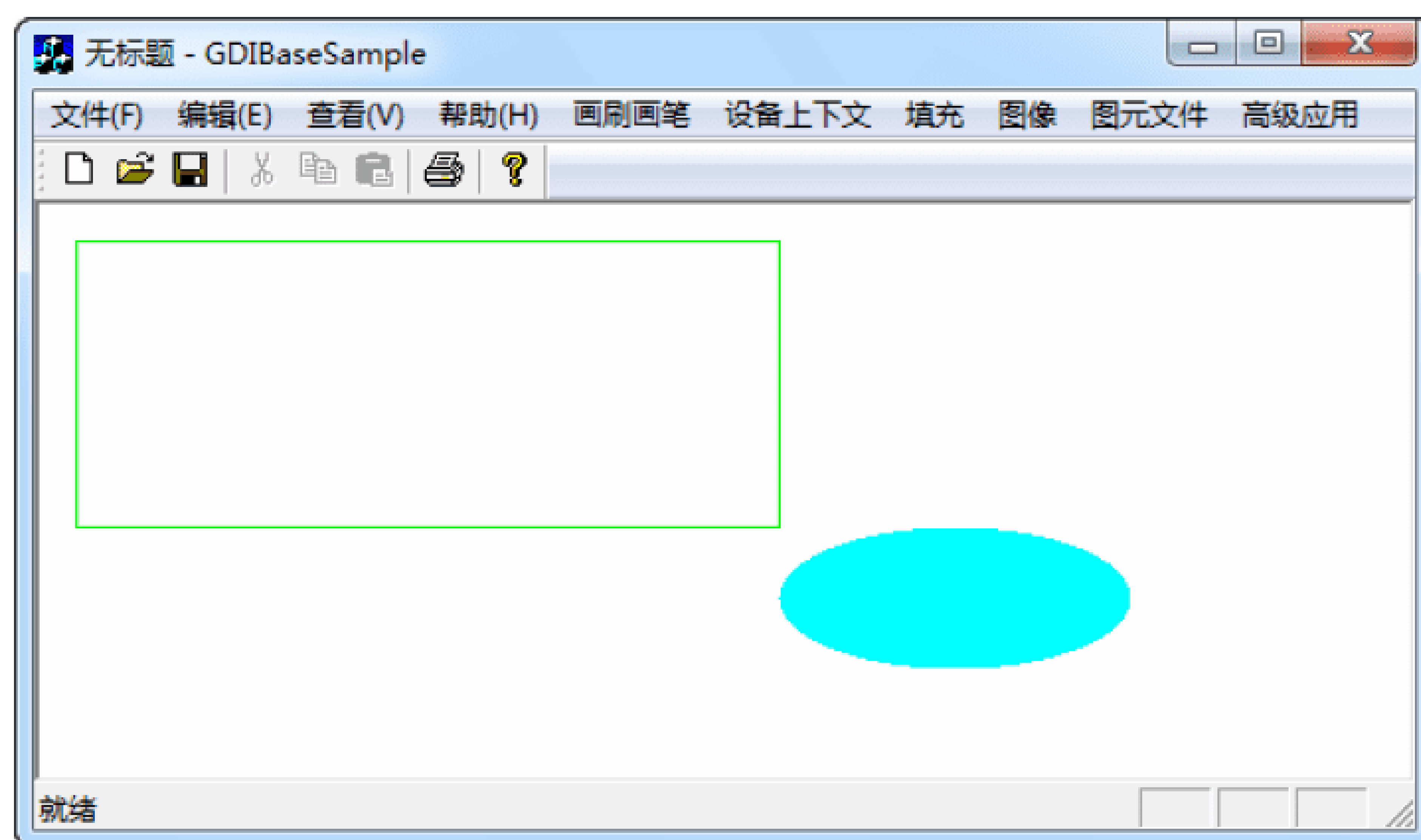


图 25-15　图元文件的保存与打开效果

25.3.13　图像居中显示

GDI+提供了 Image 类，可以用于加载、显示和处理各种格式的图像文件。本小节使用 Image 类实现图像的居中显示。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemShowpiccenter()//图像居中显示示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                       //定义 Graphics
05      status = graphics.GetLastStatus();               //获取创建状态
06      if (status != OK)
07          return;
08      Image image(L"baby.JPG");                        //装载 jpg 文件
09      status = image.GetLastStatus();                  //获取创建状态
10      if (status != OK)
11          return;
```

```
12      CRect rect;
13      GetClientRect(&rect);                            //获取工作区
14      int nLeft = 10;
15      int nTop = 20;
16      //绘制图片
17      graphics.DrawImage(&image, nLeft, nTop,
18          rect.Width()-2*nLeft, rect.Height()-2*nTop);
19  }
```

上面代码首先使用当前句柄初始化一个 Graphics 对象，使用 GetLastStatus()函数判断返回的状态值。如果返回正确，则定义 Image 对象，并使用当前目录下的 baby.jpg 文件初始化此对象。通过 GetClientRect()函数获取当前画布的大小，并定义在画布上显示图片时，要留的左右边距和上下边距，即 nLeft 和 nTop 变量指定的值。最后使用 graphics 对象的 DrawImage()函数显示图像，此函数的第二个参数和第三个参数分别用于指定显示图像的左上角的横坐标和纵坐标，第四个参数和第五个参数分别用于指定显示的图像的长和高。程序运行效果如图 25-16 所示。

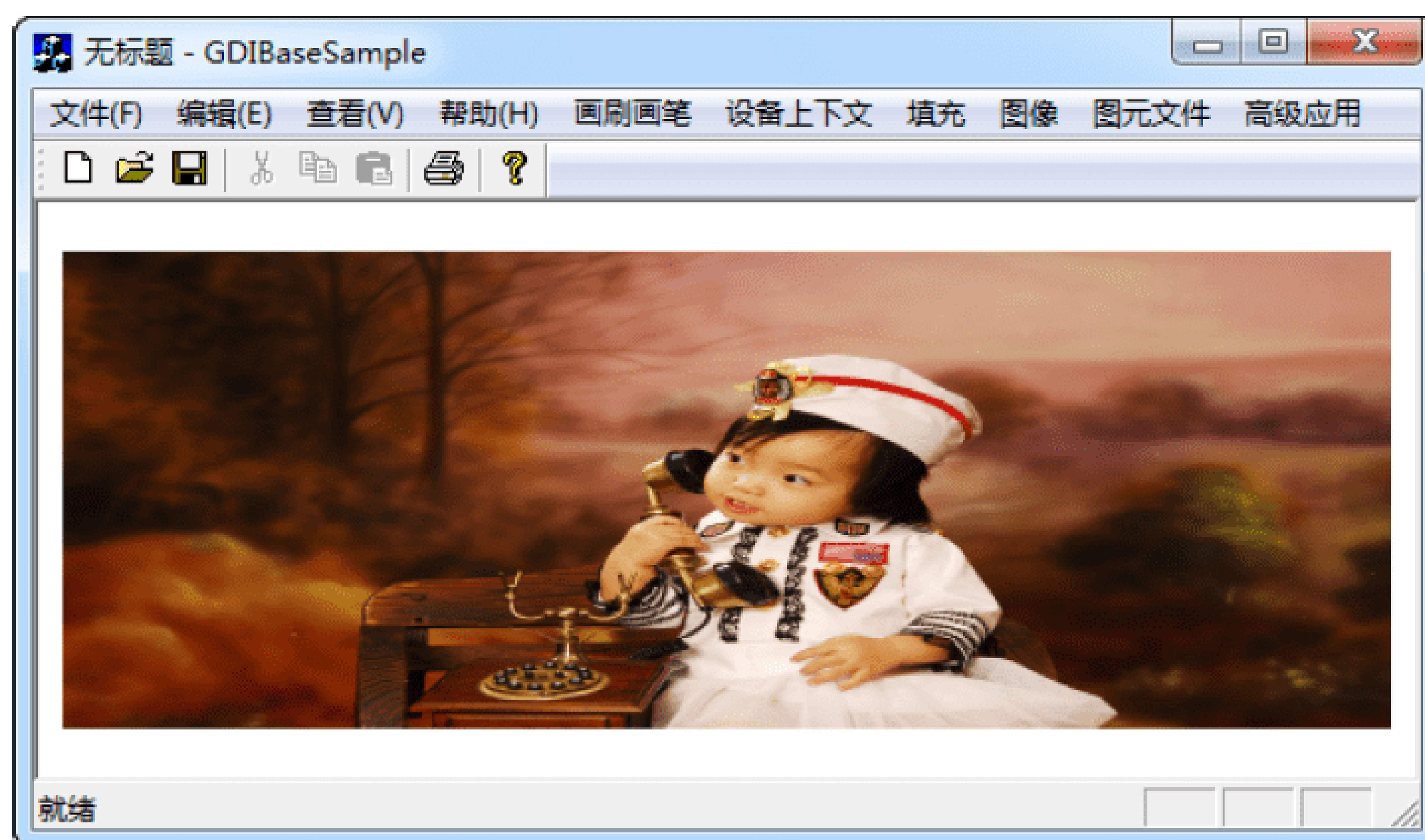


图 25-16　图像居中显示运行效果

25.3.14　图片融合效果

融合两幅图片的方法，只需要在同一区域上分别显示两幅图，在显示时将作为前景色的图片的所有像素点的颜色的透明度根据要实现的融合程度设置即可。设置的前景图片的透明度越小，则前景图片看上去越淡。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemPicturecomb()//图片融合效果示例
02  {
03      Graphics  graphics(m_hWnd);                    //定义 Graphics
04      Bitmap bg(L"girl1.jpg");                       //定义位图对象
05      int bgWidth = bg.GetWidth();
06      int bgHeight = bg.GetHeight();
07      graphics.DrawImage(&bg, 0, 0, bgWidth, bgHeight);//绘制位图
08      Bitmap fg(L"girl2.jpg");                       //定义第二幅位图对象
09      int   fgWidth = fg.GetWidth();
10      int   fgHeight = fg.GetHeight();
11      Color color, colorTemp;          //分别设置前景图中的每一个像素的透明度
```

```
12      //使用 for 循环融合两幅图片的各个对应点的像素
13      for(int iRow =0; iRow < fgHeight; iRow++)
14      {
15          for(int iColumn =0; iColumn <fgWidth; iColumn++)
16          {
17              fg.GetPixel(iColumn, iRow, &color);
18              colorTemp.SetValue(color.MakeARGB(150,color.GetRed(),
19                  color.GetGreen(),color.GetBlue()));
20              fg.SetPixel(iColumn, iRow, colorTemp);
21          }
22      }
23      //绘制融合后的图片
24      graphics.DrawImage(&fg, 0, 0, fgWidth, fgHeight);
25  }
```

上面代码中，使用 girl1.jpg 图片作为背景图片。在显示前景图片 girl2.jpg 前，首先将图片上的每个点的像素的 Alpha 值，即透明度分别设置为 150，然后将图片置于第一幅图片上显示。这样看上去就像两幅图融合在一起。程序运行效果如图 25-17 所示。

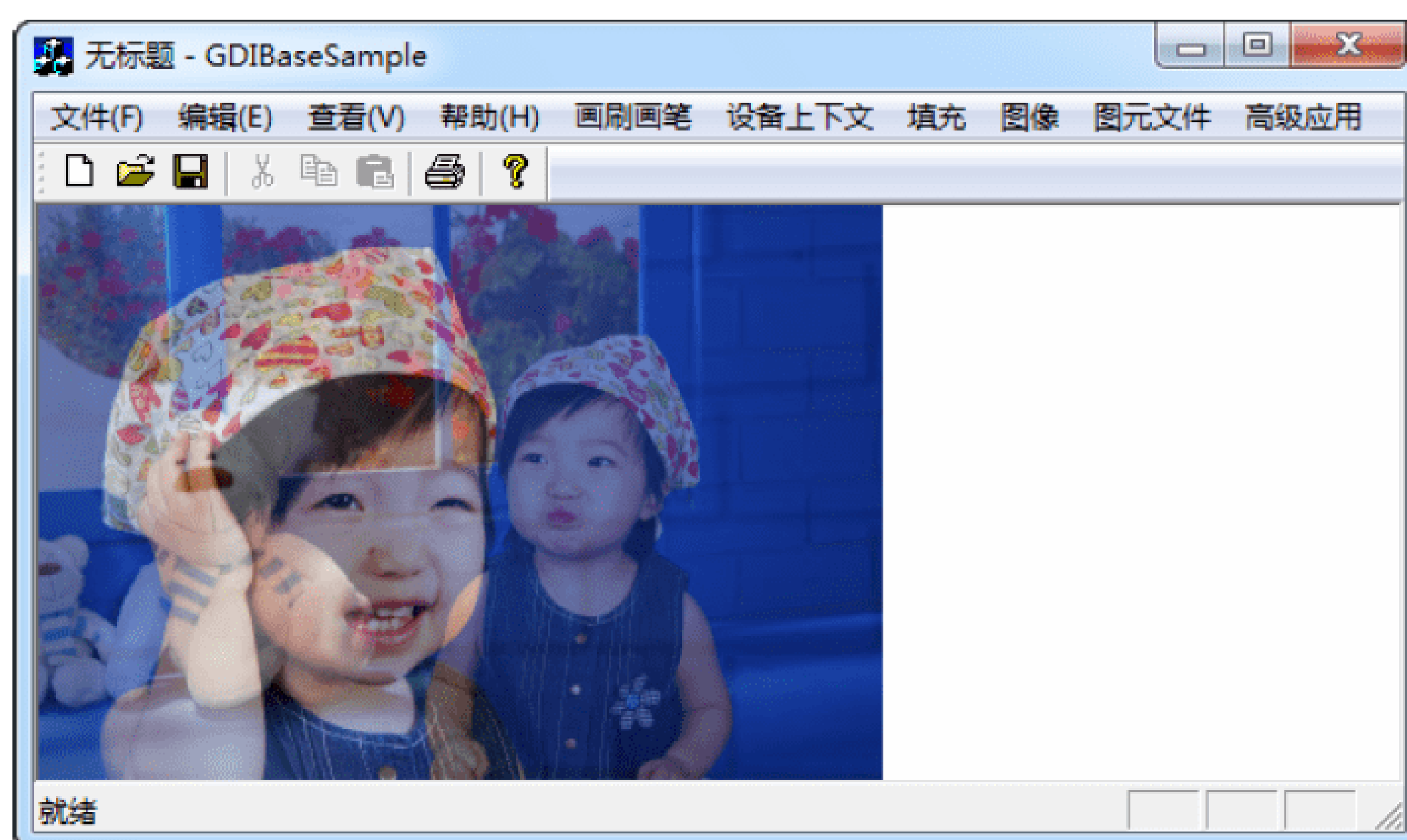


图 25-17　图片融合效果

25.3.15　保存设备上下文

在程序中，有的时候需要保存设备上下文。在使用完后，需要重新恢复原来的设备上下文。其具体处理过程代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemSavedc()     //保存设备上下文示例
02  {
03      CDC* pDC=GetDC();                            //获取设备上下文
04      CPen newPen;                                 //定义画笔对象
05      if( newPen.CreatePen( PS_SOLID, 2, RGB(255,0,255)))//创建画笔
06      {
07          int saved = pDC->SaveDC();               //保存设备上下文
08          pDC->SelectObject( &newPen );            //装载新画笔
09          CRect rect;
10          GetClientRect(&rect);                    //获取工作区
11          pDC->MoveTo(0,0);
12          pDC->LineTo(rect.Width(),rect.Height());//使用新画笔绘制直线
13          pDC->RestoreDC(saved);                   //恢复设备上下文
```

```
14            pDC->MoveTo(rect.Width(),0);
15            pDC->LineTo(0,rect.Height());//使用原来设备上下文中的画笔绘制直线
16        }
17    }
```

上面代码首先保存设备上下文，然后创建并装载新画笔，使用新画笔绘制一条工作区域的对角线。然后恢复保存的设备上下文，之后使用恢复的设备上下文绘制另一条对角线。程序运行效果如图 25-18 所示。

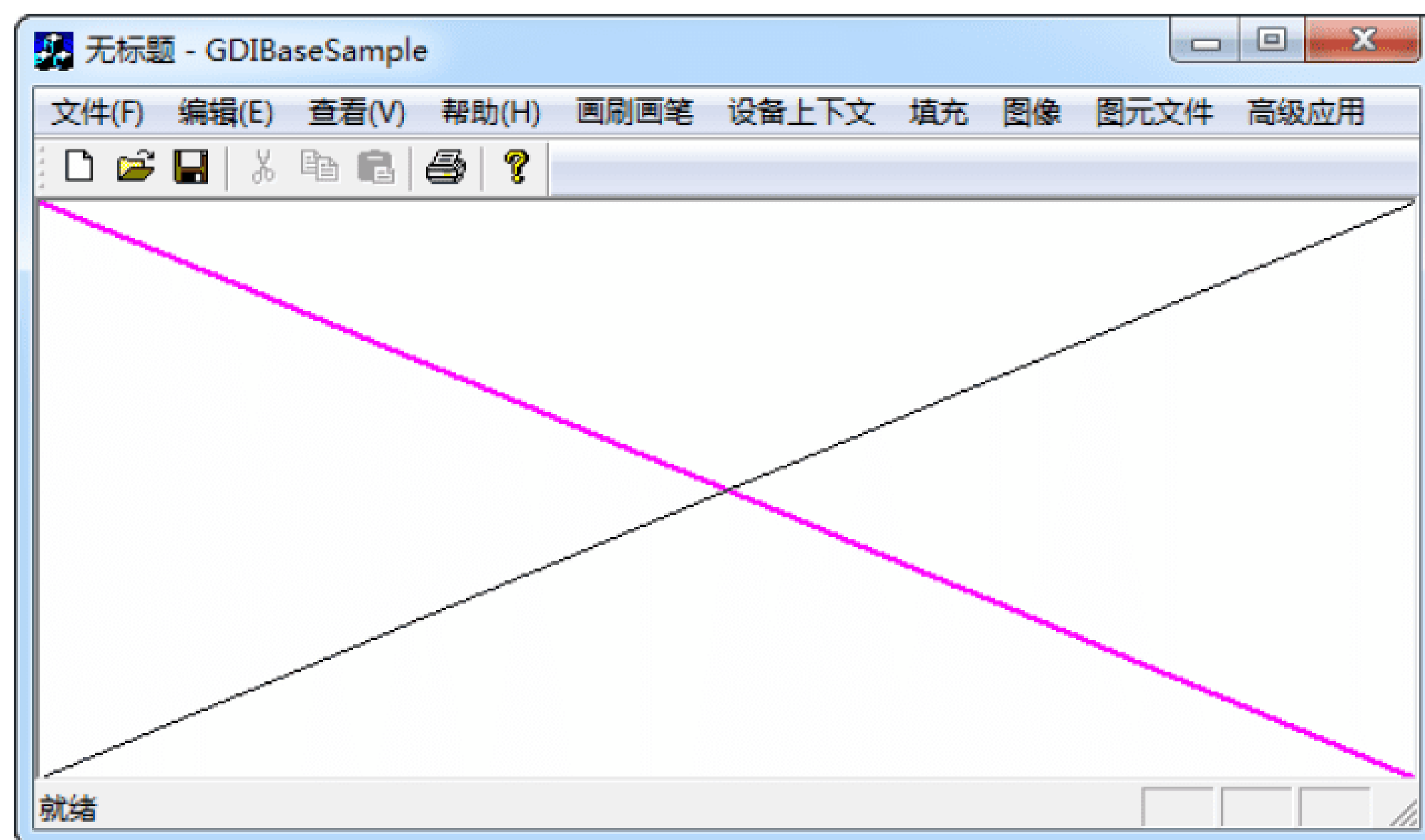


图 25-18　保存设备上下文运行效果

25.4　特殊曲线

特殊曲线的绘制是图形处理的一种典型应用。每种特殊曲线都对应特殊的数学模型，根据数学模型，将其转换成程序代码，从而可以实现绘制特殊曲线。本节将讲解使用 GDI 绘制蜗牛线和贝塞尔曲线，以及使用 GDI+绘制正弦曲线。读者通过这 3 种特殊曲线的绘制，应该掌握使用 GDI 和 GDI+绘制特殊曲线的方法，然后根据自己需要的曲线的数学模型，绘制各种特殊曲线。

25.4.1　绘制蜗牛线

本小节使用 GDI 绘制蜗牛线。蜗牛线的形状像蜗牛的外壳一样一圈套一圈，数学模型是蜗牛线上的点的坐标值符合根据指定值的正弦或余弦换算的结果。代码如下：

```
01  void CCavsSampleView::OnMenuitemWoniuline()     //绘制蜗牛线示例
02  {
03      float pi = 3.1415926f;                      //定义π的取值
04      CRect rect;
05      GetClientRect(&rect);                       //获取工作区
06      UINT width = rect.Width();
07      UINT height = rect.Height();
08      CDC* pDC = GetDC();                         //获取设备上下文
09      //根据蜗牛线的算法绘制蜗牛曲线
10      for (float x = 0;x < 10*pi; x+= pi/width)
```

```
11        {
12            if (x == 0) pDC->SetPixel(width/2, 0, RGB(255,0,0));
13            else pDC->SetPixel(width*sin(x)/x*cos(x)+width/2,
14                width*sin(x)/x*sin(x),RGB(255,0,0));
15        }
16    }
```

在上面代码中，for 循环中的变量最大值 10*pi，表示要绘制 10 圈的蜗牛线，此蜗牛线上的点的横坐标为 width*sin(x)/x*cos(x)+width/2，其中加上 width/2，表示要显示在屏幕宽度中间，纵坐标为 width*sin(x)/x*sin(x)。通过 CDC 对象的 SetPixel()函数指定蜗牛线的颜色为红色。程序运行效果如图 25-19 所示。

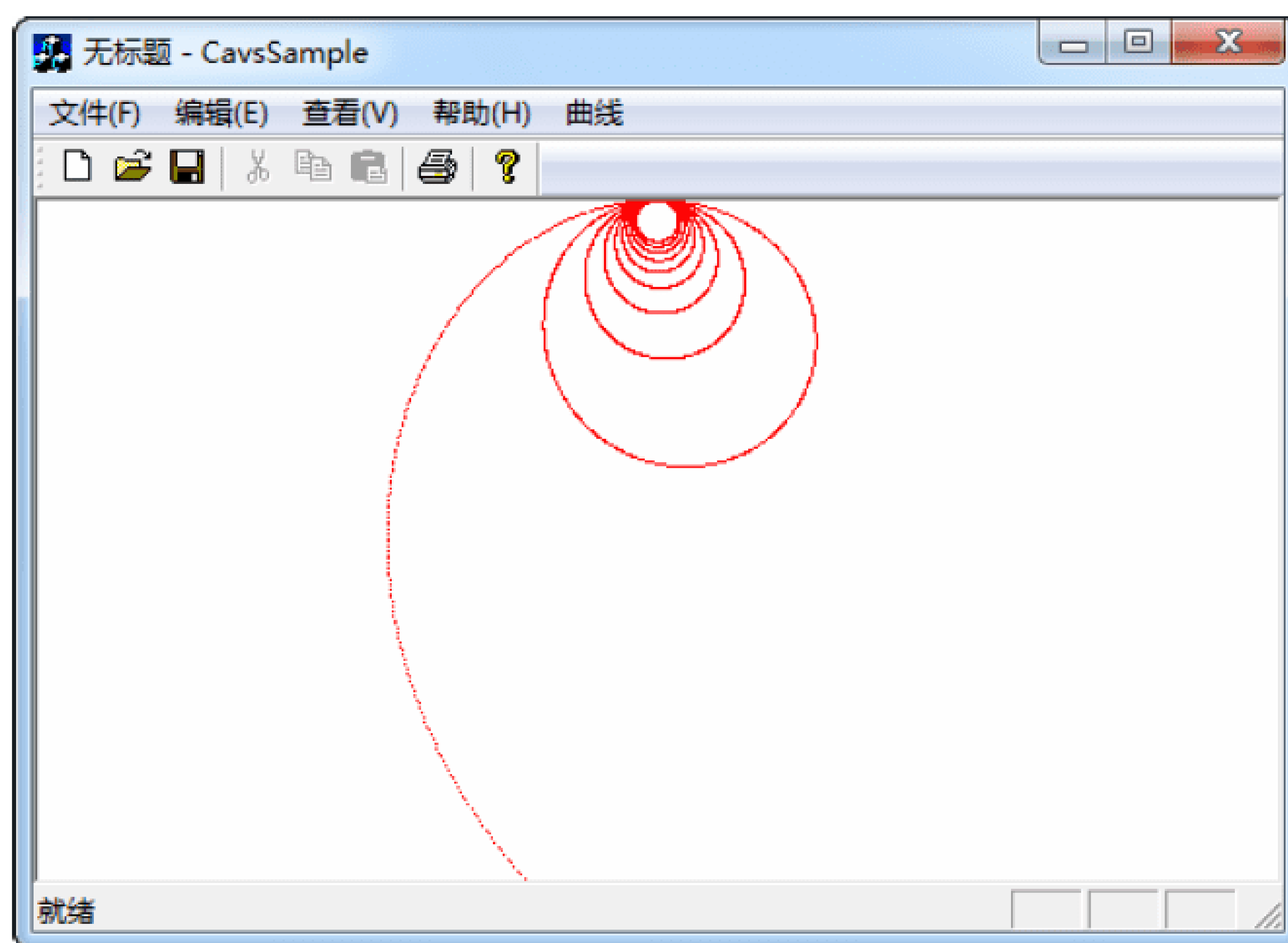


图 25-19　绘制蜗牛线运行效果

25.4.2　绘制贝塞尔曲线

贝塞尔曲线是由法国数学家贝塞尔发现的，现在广泛应用于各种绘图软件，如 Photoshop 软件中的钢笔就是贝塞尔工具。因为贝塞尔工具是将曲线作为多段线段描绘，所以使得直线和曲线都可以使用数学方法在计算机中表示，称为计算机矢量图形学的基础。每条贝塞尔曲线由 4 个结点组成，一个起点、两个控制点和一个结束点，当调整两个控制点时，会改变贝塞尔曲线的形状。也就是说，贝塞尔曲线是有方向的，其会根据路径的结点和顺序绘制曲线。

在 VC 中，使用 PolyBezier()函数可以在画布上画一条或多条贝塞尔曲线。CDC 类封装了 GDI 中的 PolyBezier 接口函数，只是将 PolyBezier 接口函数中的 CDC 参数省略了。其函数原型为：

```
BOOL PolyBezier( const POINT* lpPoints, int nCount );
```

其中，lpPoints 参数是包含结束点和曲线的控制点的 POINT 数据结构的数组。nCount 参数指定 lpPoints 数组中点的个数。此值必须是要画的曲线的数目的 3 倍加 1，因为每条贝塞尔曲线需要两个控制点和一个结束点，并且起始曲线需要增加一个起点。如果函数操作成功，则返回非 0 值，否则返回 0。

此函数使用 lpPoints 参数中指定的结点作为结束点和控制点画贝塞尔曲线。第一条贝塞尔曲线，使用第一个点到第四个点绘制。其中，第一个点为起点，第二个和第三个点为控制点，第四个点为终点；后续的贝塞尔曲线以前一条曲线的结束点和后续三个点绘制，前一条贝塞尔曲线的结束点作为下一条贝塞尔曲线的起点，接下来两个结点作为控制点，第三个结点作为结束点。函数不会使用当前位置点，也不会被 PolyBezier()函数更新。此函数使用当前画笔画贝塞尔曲线，可以根据需要，通过创建画笔，修改贝塞尔曲线的颜色和样式。以下代码显示了如何绘制贝塞尔曲线。

```
01  void CCavsSampleView::OnMenuitemDrawbezier()        //绘制贝塞尔曲线
02  {
03      CDC* pDC = GetDC();                               //获取设备上下文
04      CPoint pt[] = { CPoint(40, 72), CPoint(107, 302),
05          CPoint(329, 292),CPoint(79, 148),
06          CPoint(498, 29),CPoint(322, 201),  CPoint(176, 137) };
07      //定义贝塞尔曲线点集合
08      pDC->PolyBezier(pt, sizeof(pt)/sizeof(pt[0])); //绘制贝塞尔曲线
09  }
```

上面代码定义了 pt 数组，用于存储要绘制两条贝塞尔曲线的结点坐标。数组中的第一个结点到第四个结点是第一条贝塞尔曲线的起点、控制点和结束点；数组中的第四个结点到第七个结点是第二条贝塞尔曲线的起点、控制点和结束点。程序运行效果如图 25-20 所示。

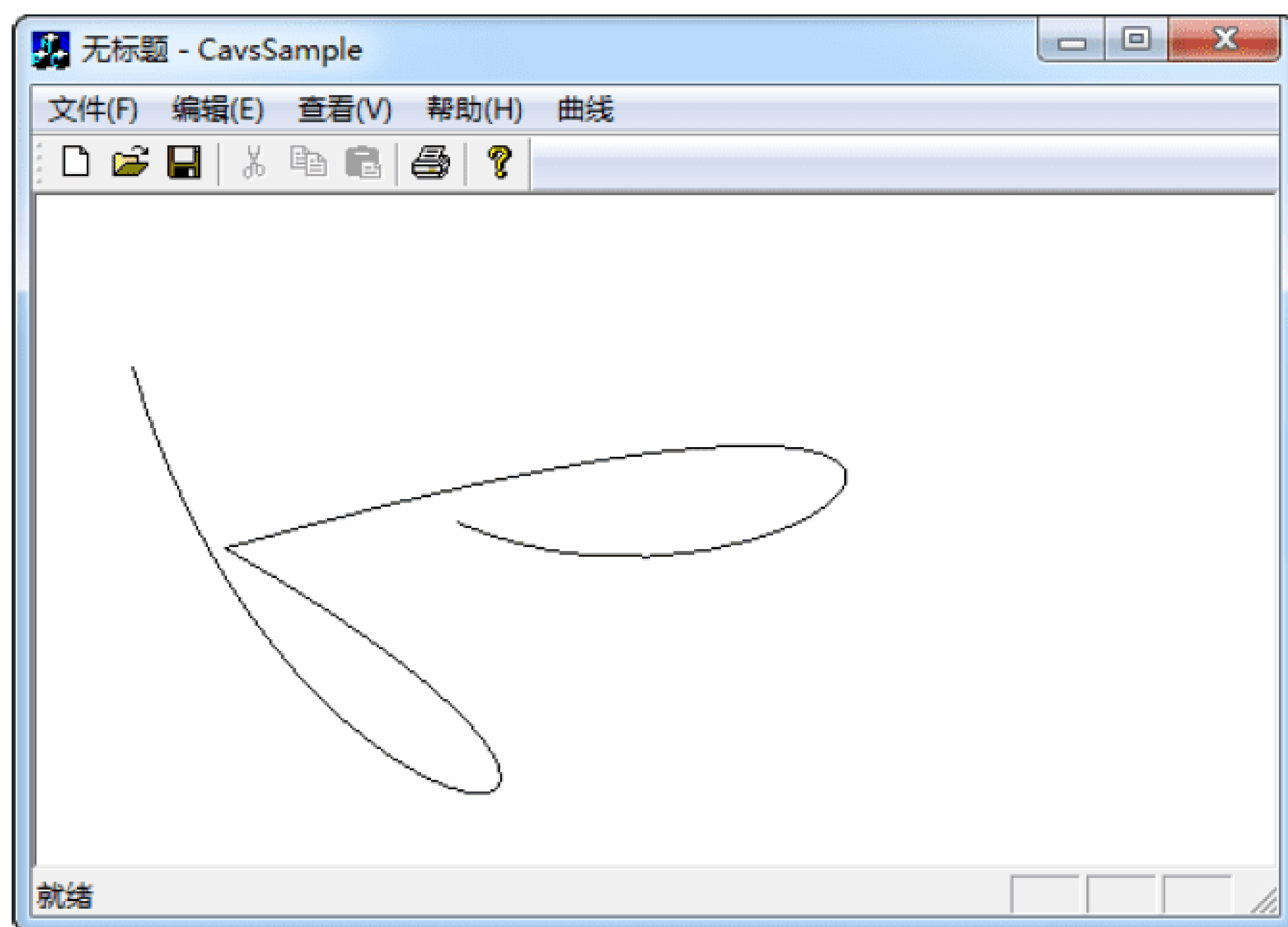


图 25-20　绘制贝塞尔曲线运行效果

25.4.3　绘制正弦曲线

本小节使用 GDI+绘制正弦曲线。正弦曲线上点的坐标符合函数 y=Asin(x)。绘制正弦曲线时，要确定正弦曲线的振幅。代码如下：

```
01  void CCavsSampleView::OnMenuitemSinline()  //绘制正弦曲线示例
02  {
03      float pi = 3.1415926f;                  //定义π的取值
04      Status status = GenericError;
05      Graphics graphics(m_hWnd);              //定义 Graphics 变量
```

```
06      status = graphics.GetLastStatus();      //获取定义结果
07      if (status != Ok)
08          return;
09      CRect rect;
10      GetClientRect(&rect);                    //获取计算工作区域
11      UINT width = rect.Width();
12      UINT height = rect.Height();
13      Pen bluePen(Color(255, 0, 0, 255), 1);  //创建画笔
14      graphics.DrawLine(&bluePen, 0, height/2, width, height/2);
15      Pen redPen(Color(255, 255, 0, 0), 2);
16      float oldX = 0.0, oldY=(width/2);
17      //绘制正弦曲线
18      for (float x = 0;x < width;x++)
19      {
20          float radian = x/width*10*pi;
21          float y = sin(radian);
22          y = (1-y) *height/2;
23          graphics.DrawLine(&redPen, oldX, oldY, x, y);
24          oldX = x;
25          oldY = y;
26      }
27  }
```

上面代码使用 DrawLine()函数在屏幕的中心绘制一条蓝色的中心线。在 for 循环中以 x 坐标，即横坐标值为计算变量，从 0 到屏幕宽。并通过正弦值和振幅计算显示的 y 值后，使用 DrawLine()函数绘制红色正弦曲线。程序运行效果如图 25-21 所示。

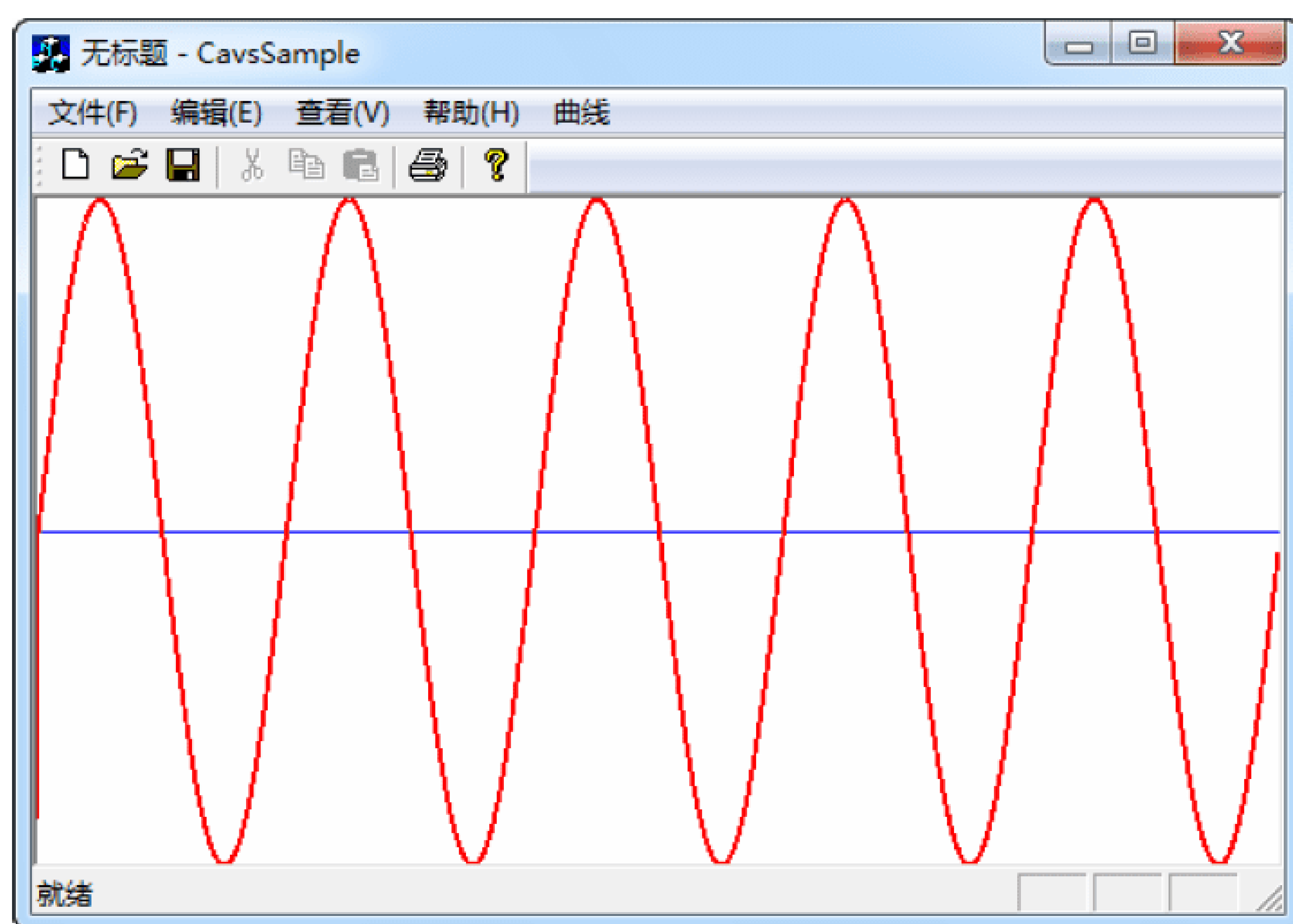


图 25-21　绘制正弦曲线运行效果图

25.5　图 像 特 效

在图像处理软件中，经常会用到一些图像特效，用于加强图像的应用效果。本节将讲解常见图像特效的实现，包括锐化、柔化、反色、灰度、浮雕效果的实现，图像翻转、图像缩放、图片剪切以及百叶窗和 3D 灰色显示图像效果的实现。

25.5.1　图像锐化处理

本小节使用拉普拉斯模板实现图像锐化处理，并在屏幕界面的左边显示原有图像，在屏幕界面的右边显示经过锐化处理后的图像。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemRuihua()  //图像锐化处理示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m hWnd);                     //定义 Graphics 变量
05      status = graphics.GetLastStatus();             //获取创建结果
06      if (status != Ok)
07          return;
08      Bitmap oldBitmap(L"girl.JPG");                 //创建 Bitmap 对象
09      status = oldBitmap.GetLastStatus();            //获取创建结果
10      if (status != Ok)
11          return;
12      UINT width = oldBitmap.GetWidth();             //获取图像尺寸
13      UINT height = oldBitmap.GetHeight();
14      Bitmap newBitmap(width, height);               //创建新的 Bitmap 对象
15      Color pixel1, pixel2, pixel;
16      //绘制原始图像
17      graphics.DrawImage(&oldBitmap, 10, 0, width, height);
18      //拉普拉斯模板
19      int Laplacian[] ={ -1, -1, -1, -1, 9, -1, -1, -1, -1 };
20      //使用锐化模板计算锐化后的图像
21      for (int x = 1; x < width - 1; x++)
22      {
23          for (int y = 1; y < height - 1; y++)
24          {
25              int r = 0, g = 0, b = 0;
26              int Index = 0;
27              for (int col = -1; col <= 1; col++)
28                  for (int row = -1; row <= 1; row++)
29                  {
30                      oldBitmap.GetPixel(x + row, y + col, &pixel);
31                      r += pixel.GetRed()* Laplacian[Index];
32                      g += pixel.GetGreen()* Laplacian[Index];
33                      b += pixel.GetBlue()* Laplacian[Index];
34                      Index++;
35                  }
36              if ( r > 255 )
37                  r = 255;
38              else if ( r < 0 )
39                  r = -r;
40              if ( g > 255 )
41                  g = 255;
42              else if ( g < 0 )
43                  g = -g;
44              if ( b > 255 )
45                  b = 255;
46              else if ( b < 0 )
47                  b = -b;
48              pixel.SetFromCOLORREF(RGB(r, g, b));
49              newBitmap.SetPixel(x - 1, y - 1, pixel);
50          }
51      }
52      //绘制锐化后的图像
53      graphics.DrawImage(&newBitmap, width+20, 0, width, height);
54  }
```

上面的代码首先载入文件并显示源文件。然后将图像由左到右、由上到下依次对每个像素点的颜色值进行拉普拉斯变化，并将变换后的颜色值进行溢出处理后，将其设置为新图像对象对应点的颜色中，最后，将新图像显示出来。程序运行效果如图 25-22 所示。



图 25-22　图像锐化处理运行效果

25.5.2　图像柔化处理

本小节使用高斯模板实现图像柔化处理，并在屏幕界面的左边显示原有图像，在屏幕界面的右边显示经过柔化处理后的图像。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemRouhua()  //图像柔化处理示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                  //定义 Graphics 变量
05      status = graphics.GetLastStatus();          //获取创建结果
06      if (status != Ok)
07          return;
08      Bitmap oldBitmap(L"girl.JPG");              //创建 Bitmap 对象
09      status = oldBitmap.GetLastStatus();         //获取创建结果
10      if (status != Ok)
11          return;
12      UINT width = oldBitmap.GetWidth();          //获取图像尺寸
13      UINT height = oldBitmap.GetHeight();
14      Bitmap newBitmap(width, height);            //创建新的 Bitmap 对象
15      Color pixel1, pixel2, pixel;
16      graphics.DrawImage(&oldBitmap, 10, 0, width, height);
17      int smoothGauss[9] = {1,2,1,2,4,2,1,2,1};   //高斯模板
18      //使用高斯模板计算柔化后的图像
19      for (int x = 1; x < width - 1; x++)
20      {
21          for (int y = 1; y < height - 1; y++)
22          {
23              int r = 0, g = 0, b = 0;
24              int Index = 0;
25              for (int col = -1; col <= 1; col++)
26                  for (int row = -1; row <= 1; row++)
27                  {
28                      oldBitmap.GetPixel(x + row, y + col, &pixel);
29                      r += pixel.GetRed()* smoothGauss[Index];
30                      g += pixel.GetGreen()* smoothGauss[Index];
```

```
31                  b += pixel.GetBlue()* smoothGauss[Index];
32                  Index++;
33              }
34              r = (r/16);
35              g = (g/16);
36              b = (b/16);
37              if ( r > 255 )
38                  r = 255;
39              else if ( r < 0 )
40                  r = -r;
41              if ( g > 255 )
42                  g = 255;
43              else if ( g < 0 )
44                  g = -g;
45              if ( b > 255 )
46                  b = 255;
47              else if ( b < 0 )
48                  b = -b;
49              pixel.SetFromCOLORREF(RGB(r, g, b));
50              newBitmap.SetPixel(x - 1, y - 1, pixel);
51          }
52      }
53      //绘制柔化后的图像
54      graphics.DrawImage(&newBitmap, width+20, 0, width, height);
55  }
```

上面的代码首先载入文件并显示源文件。然后图像由左到右、由上到下依次对每个像素点的颜色值进行高斯变化，将对应的颜色值除以 16，并将变换后的颜色值进行溢出处理后，将其设置为新图像对象对应点的颜色中，最后将新图像显示出来。程序运行效果如图 25-23 所示。



图 25-23　图像柔化处理运行效果图

25.5.3　图像反色处理

本小节使用反色图像的颜色矩阵实现图像反色处理，并在屏幕界面的左边显示原有图像，在屏幕界面的右边显示经过反色处理后的图像。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemFanse()   //图像反色示例
02  {
```

```
03      ColorMatrix colorMatrix = {
04          -1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, -1.0f,
05          0.0f, 0.0f, 0.0f,0.0f, 0.0f, -1.0f, 0.0f,
06          0.0f,0.0f, 0.0f,0.0f, 1.0f, 0.0f,
07          1.0f, 1.0f, 1.0f, 1.0f, 1.0f};       //创建反色图像的颜色矩阵
08          Status status = GenericError;
09          Graphics graphics(m_hWnd);           //定义 Graphics 变量
10          status = graphics.GetLastStatus();   //获取创建结果
11          if (status != Ok)
12              return;
13          Image image(L"girl.JPG");            //装载 Image
14          status = image.GetLastStatus();      //获取创建结果
15          if (status != Ok)
16              return;
17          UINT width = image.GetWidth();       //获取图像尺寸
18          UINT height = image.GetHeight();
19          ImageAttributes imageAttributes;
20          Rect destRect1(width+20, 10, width, height);
21          //设置图像的反色矩阵
22          imageAttributes.SetColorMatrix(&colorMatrix,
23              ColorMatrixFlagsDefault,ColorAdjustTypeBitmap);
24          graphics.DrawImage(&image, 10, 10, width, height);
25          //绘制反色后的图像
26          graphics.DrawImage(&image, destRect1, 0, 0, width,
27              height, UnitPixel,&imageAttributes);
28  }
```

上面的代码首先载入文件并显示源文件。然后定义反色图像的颜色矩阵，使用 SetColorMatrix()函数设置图像属性，最后显示经过颜色矩阵变化的图像。程序运行效果如图 25-24 所示。

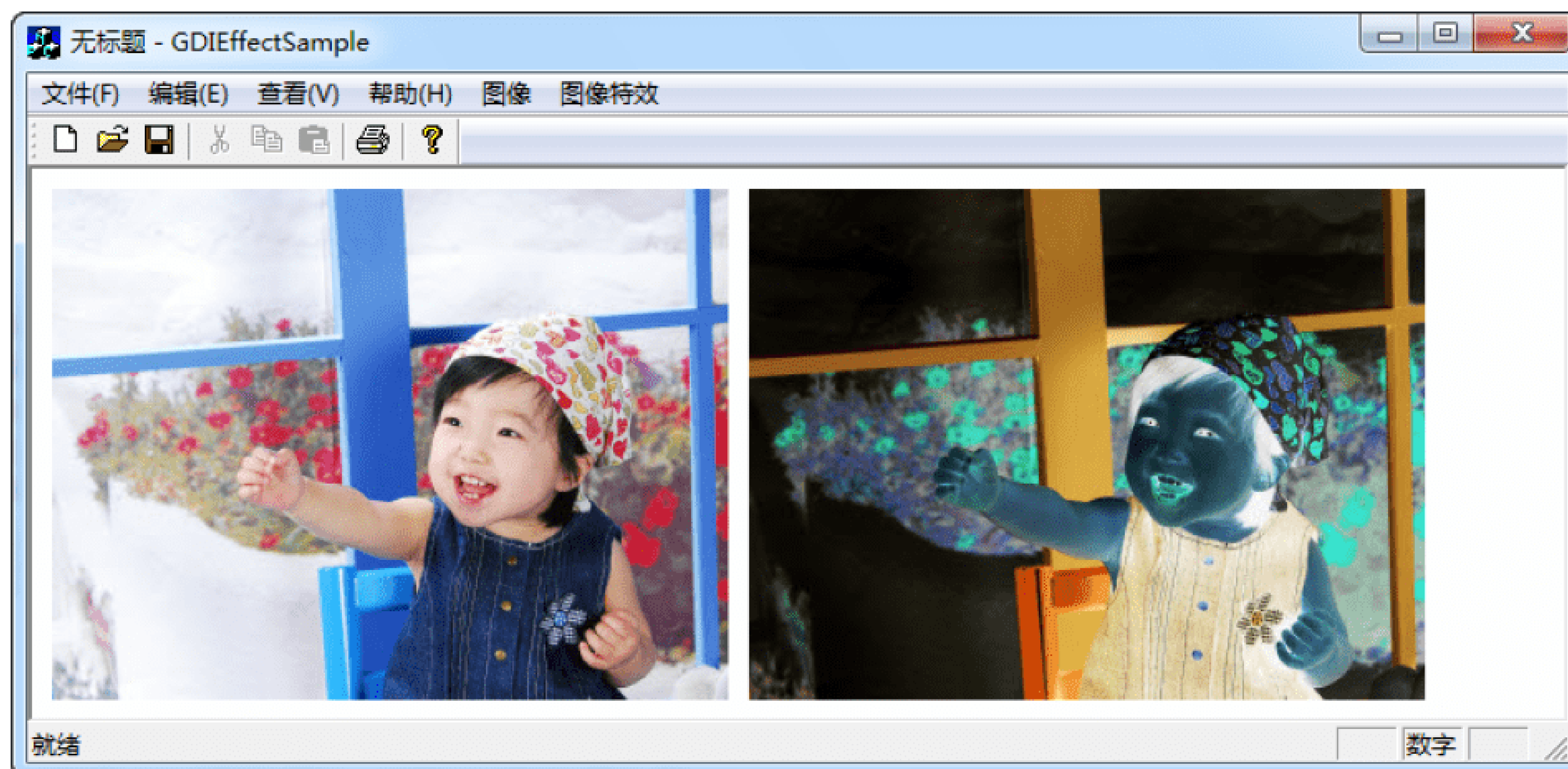


图 25-24　图像反色处理运行效果

25.5.4　图像灰度处理

本小节使用灰度图像的颜色矩阵实现图像灰度处理，并在屏幕界面的左边显示原有图像，在屏幕界面的右边显示经过灰度处理后的图像。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemHuidu()   //图像灰度示例
02  {
03      ColorMatrix colorMatrix = {
04          0.299f, 0.299f, 0.299f, 0.0f, 0.0f,
05          0.587f, 0.587f, 0.587f, 0.0f, 0.0f,
06          0.114f, 0.114f, 0.114f, 0.0f, 0.0f,
07          0.0f, 0.0f, 0.0f, 1.0f, 0.0f,0.0f,
08          0.0f, 0.0f, 0.0f, 1.0f};                  //创建灰度图像的颜色矩阵
09          Status status = GenericError;
10          Graphics graphics(m_hWnd);                 //定义 Graphics 变量
11          status = graphics.GetLastStatus();         //获取创建结果
12          if (status != Ok)
13              return;
14          Image image(L"girl.JPG");                  //装载 Image
15          status = image.GetLastStatus();            //获取创建结果
16          if (status != Ok)
17              return;
18          UINT width = image.GetWidth();             //获取图像尺寸
19          UINT height = image.GetHeight();
20          ImageAttributes imageAttributes;           //定义图像属性
21          Rect destRect1(width+20, 10, width, height);//定义区域
22          //设置图像的反色矩阵
23          imageAttributes.SetColorMatrix(&colorMatrix,
24              ColorMatrixFlagsDefault,ColorAdjustTypeBitmap);
25          graphics.DrawImage(&image, 10, 10, width, height);
26          //绘制灰度处理后的图像
27          graphics.DrawImage(&image, destRect1, 0, 0, width, height,
28              UnitPixel,&imageAttributes);
29  }
```

上面的代码首先载入文件并显示源文件。然后定义灰度图像的颜色矩阵，使用 SetColorMatrix()函数设置图像属性，最后显示经过颜色矩阵变化的图像。程序运行效果如图 25-25 所示。



图 25-25　图像灰度处理运行效果

25.5.5　图像浮雕效果

本小节使用浮雕效果颜色坐标变换实现图像浮雕效果，并在屏幕界面的左边显示原有图像，在屏幕界面的右边显示图像浮雕效果的图像。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemFudiao()  //图像浮雕效果示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                    //定义 Graphics 变量
05      status = graphics.GetLastStatus();            //获取创建结果
06      if (status != Ok)
07          return;
08      Bitmap oldBitmap(L"girl.JPG");                //创建位图对象
09      status = oldBitmap.GetLastStatus();           //获取创建结果
10      if (status != Ok)
11          return;
12      UINT width = oldBitmap.GetWidth();            //获取图像尺寸
13      UINT height = oldBitmap.GetHeight();
14      Bitmap newBitmap(width, height);              //创建新位图对象
15      Color pixel1, pixel2, pixel;
16      graphics.DrawImage(&oldBitmap, 10, 0, width, height);
17      //对新位图进行浮雕效果的像素颜色值转换
18      for (int x = 0; x < width-1; x++)
19      {
20          for (int y = 0; y < height-1; y++)
21          {
22              int r = 0, g = 0, b = 0;
23              oldBitmap.GetPixel(x, y, &pixel1);
24              oldBitmap.GetPixel(x + 1, y + 1, &pixel2);
25              r = abs(pixel1.GetRed() - pixel2.GetRed() + 128);
26              g = abs(pixel1.GetGreen() - pixel2.GetGreen() + 128);
27              b = abs(pixel1.GetBlue() - pixel2.GetBlue() + 128);
28              if (r > 255)
29                  r = 255;
30              if (r < 0)
31                  r = 0;
32              if (g > 255)
33                  g = 255;
34              if (g < 0)
35                  g = 0;
36              if (b > 255)
37                  b = 255;
38              if (b < 0)
39                  b = 0;
40              pixel.SetFromCOLORREF(RGB(r, g, b));
41              newBitmap.SetPixel(x, y, pixel);
42          }
43      }
44      //绘制经过浮雕效果转换的位图
45      graphics.DrawImage(&newBitmap, width+20, 0, width, height);
46  }
```

上面的代码首先载入文件并显示源文件。然后对颜色进行变化，分别将相邻两个点的颜色值相减，并加上 128 常数值，计算后就是浮雕效果对应的点的颜色值，最后显示浮雕效果的图像。程序运行效果如图 25-26 所示。

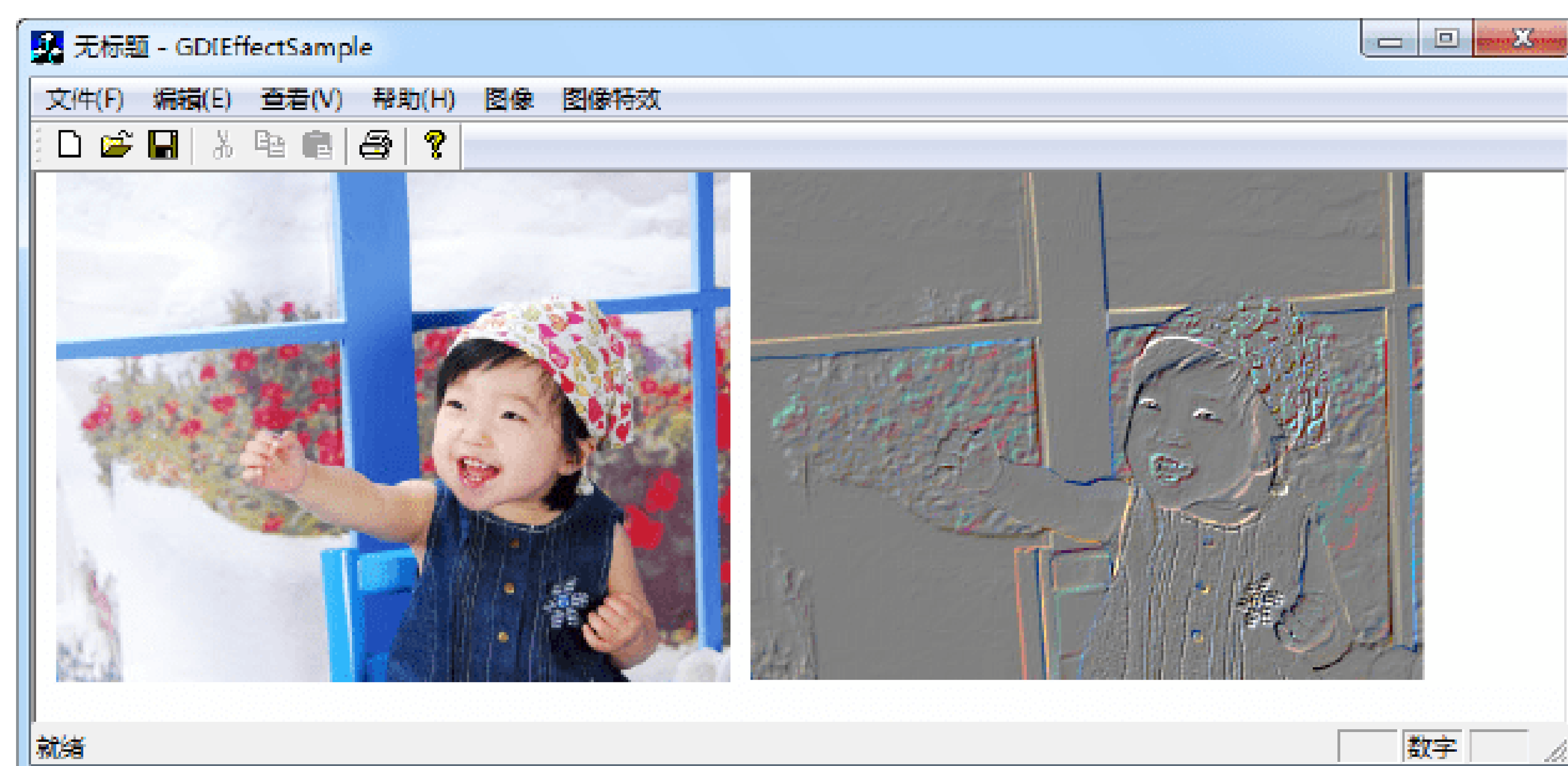


图 25-26　图像浮雕效果

25.5.6　图像翻转

本小节使用 RotateFlip()函数实现图像翻转，并在屏幕界面的左边显示原有图像，在屏幕界面的右边显示翻转后的图像。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemReserve()//图像翻转示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                        //定义 Graphics 变量
05      status = graphics.GetLastStatus();                //获取创建结果
06      if (status != Ok)
07          return;
08      Image image(L"girl.JPG");                         //创建 Image 对象
09      status = image.GetLastStatus();                   //获取创建结果
10      if (status != Ok)
11          return;
12      UINT width = image.GetWidth();                    //获取图像尺寸
13      UINT height = image.GetHeight();
14      graphics.DrawImage(&image, 10, 0, width, height);
15      Image.RotateFlip(Rotate180FlipX);                 //将图片旋转 180°
16      width = image.GetWidth();                         //获取图像尺寸
17      height = image.GetHeight();
18      //绘制翻转后的图像
19      graphics.DrawImage(&image, 20 + width, 0, width, height);
20  }
```

上面的代码首先载入文件并显示源文件。然后调用 Image 对象的 RotateFlip()函数将其翻转指定角度，本小节使用 Rotate180FlipX 表示对图像进行水平翻转。最后显示翻转后的图像。程序运行效果如图 25-27 所示。



图 25-27　图像翻转运行效果

25.5.7　图像缩放

本小节使用 DrawImage()函数的参数实现图像的缩放，并在屏幕界面的左边显示原有图像，在屏幕界面的右边显示缩小为原图像的 75%后的图像。代码如下：

```
01  void CGDIBaseSampleView::OnMenuitemResize()     //图像缩放示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                  //定义 Graphics 变量
05      status = graphics.GetLastStatus();          //获取创建结果
06      if (status != Ok)
07          return;
08      Image image(L"girl.JPG");                   //创建 Image 对象
09      status = image.GetLastStatus();             //获取创建结果
10      if (status != Ok)
11          return;
12      UINT width = image.GetWidth();              //获取图像尺寸
13      UINT height = image.GetHeight();
14      graphics.DrawImage(&image, 0, 0, width, height);//绘制原始图像
15      //绘制缩放后的图像
16      graphics.DrawImage(&image, width,
17                          0, 0.75 * width, 0.75 * height);
18  }
```

上面的代码首先载入文件并显示源文件。然后调用 DrawImage()函数，通过第三个参数和第四个参数来指定要绘制的图像的大小，此处为原图像的 75%大小。程序运行效果如图 25-28 所示。



图 25-28　图像缩放运行效果

25.5.8　图片剪切

本小节使用 DrawImage()函数的参数实现图像剪切，并在屏幕界面的左边显示原有图像，在屏幕界面的右边显示剪切后的图像。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemCut()      //图片剪切示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                  //定义 Graphics 变量
05      status = graphics.GetLastStatus();          //获取创建结果
06      if (status != Ok)
07          return;
08      Image image(L"girl.JPG");                   //创建 Image 对象
09      status = image.GetLastStatus();             //获取创建结果
10      if (status != Ok)
11          return;
12      UINT width = image.GetWidth();
13      UINT height = image.GetHeight();            //获取图像尺寸
14      Rect destRect1(width+20, 20, 30, 30);       //小范围
15      Rect destRect2(width+20, 100, 120, 120);    //大范围
16      graphics.DrawImage(&image, 0, 0, width, height);//绘制原始图像
17      //绘制缩小剪切
18      graphics.DrawImage(&image, destRect1, 87, 123, 35,
19                         35, UnitPixel);
20      graphics.SetInterpolationMode(
21                         InterpolationModeHighQuality Bilinear);
22      //绘制放大剪切
23      graphics.DrawImage(&image, destRect2, 87, 123, 35,
24                        35, UnitPixel);
25  }
```

上面的代码首先载入文件并显示源文件。然后调用 DrawImage()函数，通过第二个参数指定剪切的图像显示的范围大小，第三个参数和第四个参数指定要绘制的图像在源图像的位置点，第五个参数和第六个参数用来指定要从源图像上剪切的图像的大小。程序运行效果如图 25-29 所示。



图 25-29　图像剪切运行效果

25.5.9　图片马赛克效果

本小节使用 DrawImage()函数随机显示一块图像实现图片马赛克效果。随机选取的块通过 rand()随机函数生成块代号，通过代码计算块所表示的区域，并显示出此区域上的图

像，直到所有的块都显示完成，整个图片马赛克的效果就完成了。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemMasaike() //图片马赛克效果示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                    //定义 Graphics 变量
05      status = graphics.GetLastStatus();            //获取创建结果
06      if (status != Ok)
07          return;
08      graphics.Clear(Color.White);                  //清除画布上的白色像素
09      Image image(L"girl.JPG");                     //创建 Image 对象
10      status = image.GetLastStatus();               //获取创建结果
11      if (status != Ok)
12          return;
13      UINT width = image.GetWidth();
14      UINT height = image.GetHeight();              //获取图像尺寸
15      int dw = width / 50;                          //定义马赛克块的大小
16      int dh = height / 50;
17      int points[2500]={0};
18      int nCount=0;
19      //使用 while 循环输出马赛克效果
20      while(nCount < 2500)
21      {
22          int index = rand()%2500;
23          if (points[index] == 0)
24          {
25              nCount ++;
26              points[index] = 1;
27              int m = index/50;
28              int n = index%50;
29              Rect destRect(dw*m, dh*n, (m+1)*dw, (n+1)*dh);
30              //绘制马赛克图像块
31              graphics.DrawImage(&image, destRect, dw*m, dh*n,
32                  (m+1)*dw,(n+1)*dh, UnitPixel);
33          }
34          Sleep(10);
35      }
36  }
```

上面的代码首先载入文件，通过 while 循环，随机选取要绘制的图像块，并记录图像块是否绘制过，直到所有的块绘制完毕。程序运行效果如图 25-30 所示。



图 25-30　图片马赛克运行效果

25.5.10　垂直百叶窗显示图片

本小节通过定时、分部分显示图像的方式实现水平百叶窗显示图片。将整个图像分成 30 次显示，每次显示相隔相等距离的竖条图案，并在每次显示之间停留 10 毫秒，这样效果就像水平百叶窗一样。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemHbaiye()//垂直百叶窗显示图片示例
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                //定义 Graphics 变量
05      status = graphics.GetLastStatus();        //获取创建结果
06      if (status != Ok)
07          return;
08      graphics.Clear(Color.White);              //清除画布上的白色像素
09      Image image(L"girl.JPG");                 //创建 Image 对象
10      status = image.GetLastStatus();           //获取创建结果
11      if (status != Ok)
12          return;
13      UINT width = image.GetWidth();
14      UINT height = image.GetHeight();          //获取图像尺寸
15      int nPixes = 30;
16      int nNum = width/nPixes;                  //计算水平百叶窗的个数
17      for (int i = 0;i <nPixes; i++)
18      {
19          for(int j=0;j<nNum;j++)
20          {
21              //分别扫描每条
22              Rect destRect1(j*nPixes+i, 0, 1, height);
23              //绘制每条百叶条的图像
24              graphics.DrawImage(&image, destRect1,
25                  j*nPixes+i, 0, 1, height,UnitPixel);
26          }
27          Sleep(10);
28      }
29  }
```

上面的代码首先载入文件，然后通过两个嵌套的 for 循环，使用 DrawImage()函数分次显示相隔一定距离的图像条。程序运行效果如图 25-31 所示。

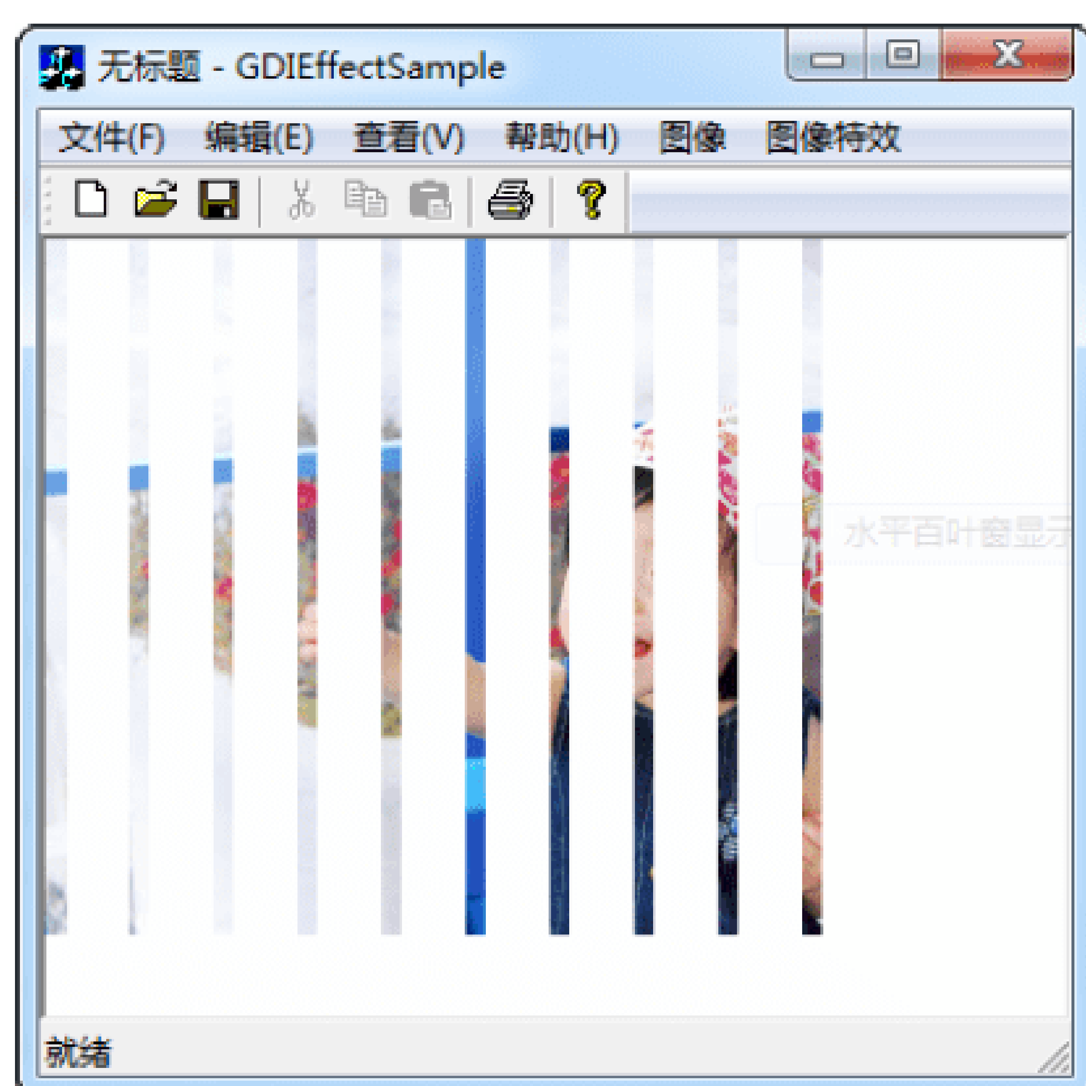


图 25-31　垂直百叶窗显示图片运行效果

25.5.11　水平百叶窗显示图片

本小节通过定时、分部分显示图像的方式实现垂直百叶窗显示图片。将整个图像分成 30 次显示，每次显示相隔相等距离的横条图案，并在每次显示之间停留 50 毫秒，这样效果就像垂直百叶窗一样。代码如下：

```
01  void CGDIEffectSampleView::OnMenuitemVbaiye()
02  {
03      Status status = GenericError;
04      Graphics graphics(m_hWnd);                  //定义 Graphics 变量
05      status = graphics.GetLastStatus();          //获取创建结果
06      if (status != Ok)
07          return;
08      graphics.Clear(Color.White);                //清除画布上的白色像素
09      Image image(L"girl.JPG");                   //创建 Image 对象
10      status = image.GetLastStatus();             //获取创建结果
11      if (status != Ok)
12          return;
13      UINT width = image.GetWidth();
14      UINT height = image.GetHeight();            //获取图像尺寸
15      int nPixes = 15;
16      int nNum = height /nPixes;                  //计算垂直百叶窗的个数
17      or (int i = 0;i <nPixes; i++)
18      {
19          for(int j=0;j<nNum;j++)
20          {//分别扫描每条
21              Rect destRect1(0, j*nPixes+i, width, 1);
22              //绘制每条百叶条的图像
23              graphics.DrawImage(&image, destRect1,
24                  0, j*nPixes+i, width, 1,UnitPixel);
25          }
26          Sleep(50);
27      }
28  }
```

上面的代码首先载入文件，然后通过两个嵌套的 for 循环，使用 DrawImage()函数分次显示相隔一定距离的横向图像条。程序运行效果如图 25-32 所示。



图 25-32　水平百叶窗显示图片运行效果

25.6　图　像　控　制

本节介绍有关图像控制的实例，通过本节的学习，读者应该可以掌握图像控制的基本方法，并能根据需要完成自定义的图像控制功能。本节主要讲解了如下几个示例：在图片上绘制线条和网格、创建最顶层窗体、如何在视图中拖动图片、如何实现屏幕截图并保存、获取图像 RGB 值、显示 Word 艺术字、渐隐渐现图像的实现以及区域的使用。

25.6.1　在图片上绘制线条

使用 GDI+技术要在图片上绘制线条，只需要先在设备上下文中使用 DrawImage()函数显示图片，然后在图片上使用 DrawLine()函数绘制需要的线条就可以了。代码如下：

```
01  void CGDIControlView::OnMenuitemDrawlineonimage()//在图片上绘制线条示例
02  {
03      Graphics graphics(m_hWnd);                         //定义 Graphics 变量
04      Image image(L"baby.JPG");                          //创建 Image 对象
05      UINT width = image.GetWidth();
06      UINT height = image.GetHeight();                   //获取图像尺寸
07      //计算线条区域
08      Rect destRect(width+20, 10, width, height);
09      graphics.DrawImage(&image, 10, 10, width, height);//绘制原始图像
10      //绘制目标区域
11      graphics.DrawImage(&image, destRect, 0, 0, width,
12          height, UnitPixel);
13      Pen pen(Color(255, 255, 0, 0), 3);                     //定义画笔
14      int nLineCount = 18;
15      int h = height/(nLineCount-1);                         //计算画笔的宽度
16      for (int i = 0;i <nLineCount; i++)
17          //绘制线条
18          graphics.DrawLine(&pen, width+20, h*i+10,
19          width*2 + 20, h*i+10);
20  }
```

上面代码首先创建 Graphics 对象和 Image 对象，并对其进行初始化。然后使用 DrawImage()函数并列显示两幅原图。最后，定义红色画笔，在第二幅图上依次绘制 18 条横线。程序运行效果如图 25-33 所示。



图 25-33　在图片上绘制线条运行效果

25.6.2　在图片上绘制网格

使用 GDI+技术要在图片上绘制网格，与在图片上绘制线条的方法是一样的。只需要先在设备上下文中使用 DrawImage()函数显示图片，然后在图片上使用 DrawLine()函数绘制相交的线条就形成图片上的网格效果。代码如下：

```
01  void CGDIControlView::OnMenuitemDrawnetlineonimage()
02  {
03      Graphics graphics(m_hWnd);                       //定义 Graphics 变量
04      Image image(L"baby.JPG");                        //创建 Image 对象
05      UINT width = image.GetWidth();
06      UINT height = image.GetHeight();                 //获取图像尺寸
07      Rect destRect(width+20, 10, width, height);
08      graphics.DrawImage(&image, 10, 10, width, height);
09      graphics.DrawImage(&image, destRect, 0, 0,
10          width, height, UnitPixel);
11      //绘制目标区域
12      Pen pen(Color(255, 255, 0, 0), 3);
13      int nXCount = 18;
14      int nYCount  = 18;
15      //绘制横线
16      int h = height/(nXCount-1);
17      for (int i = 0;i <nXCount; i++)
18          graphics.DrawLine(&pen, width+20, h*i+10,
19          width*2 + 20, h*i+10);
20      //绘制纵线
21      int w = width/(nYCount-1);
22      for (i = 0;i <nYCount; i++)
23          graphics.DrawLine(&pen, w*i+20+width, 10,
24          w*i+20+width, 10+height);
25  }
```

上面代码首先创建 Graphics 对象和 Image 对象，并对其进行初始化，然后使用 DrawImage()函数并列显示两幅原图。最后，定义红色画笔，在第二幅图上依次绘制 18 条横线和 18 条纵线。程序运行效果如图 25-34 所示。



图 25-34　在图片上绘制网格运行效果

25.6.3　打开高颜色质量图像

使用 GDI+的 Graphics 对象的 SetInterpolationMode()函数可以设置打开图像时图像的颜

色质量。有效取值如下。

- InterpolationModeNearestNeighbor：此方式是质量最差的模式。
- InterpolationModeBilinear：双线性模式。
- InterpolationModeHighQualityBilinear：高质量双线性模式。
- InterpolationModeBicubic：双三次换算模式。
- InterpolationModeHighQualityBicubic：高质量双三次换算模式，此种模式是颜色质量最好的一种模式。

以下代码使用 InterpolationModeHighQualityBicubic 颜色质量模式打开高颜色质量图像。

```
01  void CGDIControlView::OnMenuitemOpenhighimage()//打开高颜色质量图像示例
02  {
03      Graphics graphics(m_hWnd);                     //定义 Graphics 变量
04      Image image(L"baby.JPG");                      //创建 Image 对象
05      UINT width = image.GetWidth();
06      UINT height = image.GetHeight();               //获取图像尺寸
07      graphics.SetInterpolationMode
08            (InterpolationModeHighQualityBicubic);
09      graphics.DrawImage(&image, 10, 10, width, height);
10      graphics.SetInterpolationMode(InterpolationModeNearestNeighbor);
11      //绘制高质量图像
12      graphics.DrawImage(&image, width + 20, 10, width, height);
13  }
```

上面代码首先创建 Graphics 对象和 Image 对象，并对其进行初始化。然后调用 SetInterpolationMode()函数设置颜色质量模式为 InterpolationModeHighQualityBicubic，显示图像。调用 SetInterpolationMode()函数设置颜色质量模式为 InterpolationModeNearest-Neighbor，显示图像。从而比较颜色质量高的设置和颜色质量低的设置，查看这两种情况的比较效果。程序运行效果如图 25-35 所示。



图 25-35　打开高颜色质量图像运行效果

25.6.4　创建最顶层窗体

要创建最顶层窗体，需要在创建窗体的 OnCreate()函数中调用 SetWindowLong()函数和 SetLayeredWindowAttributes()函数进行设置，并使用 SetWindowPos()函数显示对话框。

代码如下：

```
01  typedef BOOL (WINAPI *SETLAYERFUNC)(HWND,COLORREF,BYTE,DWORD);
02  int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
03  {
04      SetWindowLong(this->GetSafeHwnd(),GWL_EXSTYLE,
05          GetWindowLong(this->GetSafeHwnd(),GWL_EXSTYLE)^0x80000);
06      HINSTANCE hInst = LoadLibrary("User32.DLL");//装载 User32.dll
07      if(hInst)
08      {
09          SETLAYERFUNC func = NULL;
10          //获取 SetLayeredWindowAttributes()函数指针
11          func=(SETLAYERFUNC)GetProcAddress(hInst,
12              "SetLayeredWindowAttributes");
13          if(func)
14              func(this->GetSafeHwnd() ,0, 200, 2);//调用函数
15          else
16              MessageBox("装载函数失败");
17          FreeLibrary(hInst);                         //释放动态链接库
18      }
19      //发送窗体置顶的消息
20      ::SetWindowPos(this->GetSafeHwnd(),HWND_TOPMOST,
21          0, 0, 300, 180, SWP_SHOWWINDOW);
22      return 0;
23  }
```

上面代码首先调用 SetWindowLong()函数设置对话框具有扩展属性。由于 API 不提供 SetLayeredWindowAttributes()接口函数，因此，使用 LoadLibrary()函数装载 User32.DLL 文件，并通过 GetProcAddress()函数获取 SetLayeredWindowAttributes 接口函数的指针，通过指针调用此函数，设置窗体的透明度为 200，并释放文件。最后调用 SetWindowPos()函数显示对话框。程序运行效果如图 25-36 所示。



图 25-36　创建最顶层窗体运行效果

25.6.5　在视图中拖动图片

要实现在视图中拖动图片，分为 3 个步骤。

（1）需要处理按钮命令事件。此部分的功能是，设置类变量 OperType 的值为 3，表示

当前处于拖动图片的命令方式，并且在原点处显示初始图片。代码如下：

```
01  void CGDIControlView::OnMenuitemDrag()      //在视图中拖动图片示例
02  {
03      OperType = 3;
04      Graphics graphics(m hWnd);              //定义 Graphics 变量
05      graphics.Clear(Color.White);            //清除图片中的白色像素点
06      graphics.SetSmoothingMode(SmoothingModeAntiAlias);
07      Bitmap image(L"baby.JPG");              //创建位图对象
08      UINT width = image.GetWidth();
09      UINT height = image.GetHeight();        //获取图像尺寸
10      graphics.DrawImage(&image, 0, 0, width, height);//绘制原始图片
11      rect.left = 0;                          //记录当前工作区
12      rect.top = 0;
13      rect.right = width;
14      rect.bottom = height;
15  }
```

（2）需要处理鼠标按下事件。在鼠标按下事件的处理函数中，首先判断是否处于拖动图片状态下，如果是，则判断当前鼠标按下的点的坐标是否属于图片的范围，如果是，则设置类变量 bSelectedImage 的值为 true，表示已经选择图片，并将当前鼠标的位置点保存在 Point 类型的类变量 forePoint 中。代码如下：

```
01  //鼠标按下的处理函数
02  void CGDIControlView::OnLButtonDown(UINT nFlags, CPoint point)
03  {
04      …
05          if(OperType == 3)
06          {
07              Graphics graphics(m_hWnd);    //定义 Graphics 变量
08              //获取区域
09              Region region(Rect(rect.left, rect.top,
10                  rect.right, rect.bottom));
11              PointF backPoint((float)point.x, (float)point.y);
12              //判断单击点是否在图像区域内
13              if(region.IsVisible(backPoint, &graphics))
14              {
15                  bSelectedImage = true;    //如果是，设置选择图片变量为 true
16                  forePoint.x = point.x;    //记录信息点位置
17                  forePoint.y = point.y;
18              }
19          }
20          …
21  }
```

（3）需要处理鼠标抬起事件。首先判断是否处于拖动图片状态下，如果是，则判断类变量 bSelectedImage 是否为真，如果是真，表示此次鼠标抬起事件对应的鼠标按下事件选择了图片，则将当前点和先前记录在 forePoint 变量中的点结合起来计算图像移动的偏移量，并移动图片。代码如下：

```
01  //鼠标按键抬起的处理函数
02  void CGDIControlView::OnLButtonUp(UINT nFlags, CPoint point)
03  {
04      …
05      else if (OperType == 3)
06      {
```

```
07          //拖动图片
08          if (bSelectedImage)              //如果当前选择了图片
09          {
10              bSelectedImage = false;
11              Graphics graphics(m_hWnd);   //定义 Graphics 变量
12              graphics.Clear(Color.White); //清除图片中的白色像素点
13              Image image(L"baby.JPG");    //创建 Image 对象
14              UINT width = image.GetWidth();
15              UINT height = image.GetHeight(); //获取图像尺寸
16              //计算鼠标拖动后的图像的位置
17              int offX = point.x - forePoint.x;
18              int offY = point.y - forePoint.y;
19              rect.left += offX;
20              rect.top += offY;
21              //在新位置点绘制图片
22              graphics.DrawImage(&image, rect.left, rect.top,
23                                  width, height);
24          }
25      }
26      …
27  }
```

25.6.6　屏幕截图

屏幕截图的原理就是将屏幕抓下来，将其复制到位图对象上，再由位图对象在屏幕画布上显示。本小节中以 GDI+和 GDI 结合的方式，讲述如何实现屏幕截取。代码如下：

```
01  void CGDIControlView::OnMenuitemScreencut()    //屏幕截图示例
02  {
03      int cx = GetSystemMetrics(SM_CXSCREEN);     //获取屏幕分辨率
04      int cy = GetSystemMetrics(SM_CYSCREEN);
05      HDC hScrDC = CreateDC("DISPLAY",NULL,NULL,NULL);
06      //创建显示器设备上下文
07      Graphics graphics1(hScrDC);                 //定义 Graphics 变量
08      Bitmap bitmap(cx, cy, &graphics1);          //定义位图变量
09      Graphics graphics2(&bitmap);                //定义 Graphics 变量
10      HDC dc1 = graphics1.GetHDC();               //获取设备上下文句柄
11      HDC dc2 = graphics2.GetHDC();
12      BitBlt(dc2,0,0,cx,cy,dc1,0,0,13369376);     //绘制屏幕截图
13      graphics1.ReleaseHDC(dc1);                  //释放设备上下文
14      graphics2.ReleaseHDC(dc2);
15      Graphics graphics(m_hWnd);                  //定义窗体 Graphics 变量
16      UINT width = bitmap.GetWidth();
17      UINT height = bitmap.GetHeight();           //获取图像尺寸
18      //将屏幕截图绘制在位图上
19      graphics.DrawImage(&bitmap, 0, 0, width, height);
20  }
```

上面代码首先调用 GetSystemMetrics()函数获取当前屏幕的大小范围。然后调用 CreateDC()函数创建屏幕 DISPLAY 画布，并使用此画布初始化 Graphics 对象，将 Bitmap 对象绑定到新 Graphics 对象。接下来，获取 Graphics 对象的 HDC 画布句柄，使用 BitBlt()函数复制屏幕内容到位图对象中。最后将位图中的内容绘制在当前程序中的画布上。程序运行效果如图 25-37 所示。

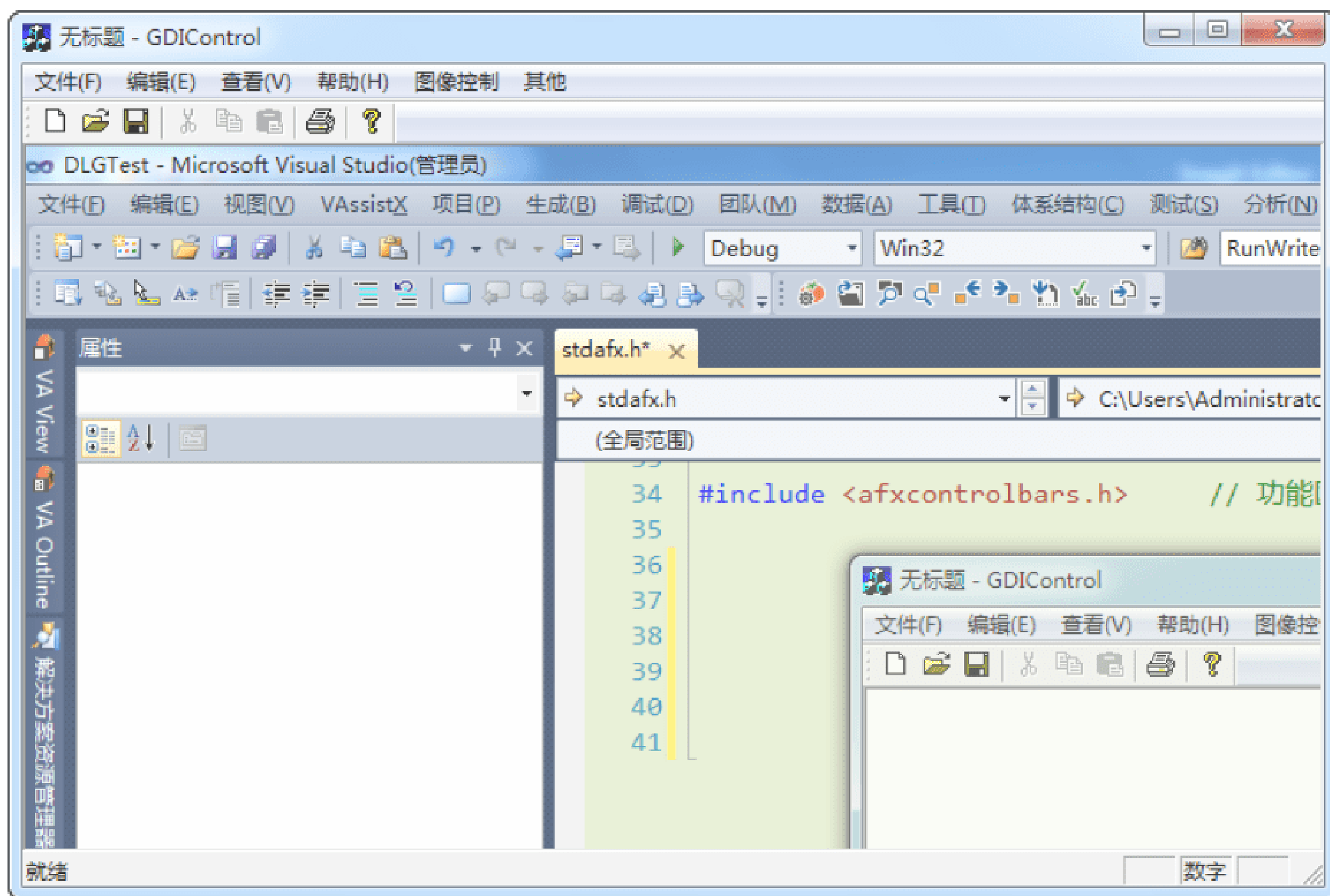


图 25-37　屏幕截图运行效果

25.6.7　保存屏幕图像到剪贴板

保存屏幕图像到剪贴板，与屏幕截图的方法是一样的。区别在于，此处只将截取到的屏幕放置在剪贴板中，而不显示在输出设备上。代码如下：

```
01  void CGDIControlView::OnMenuitemScreencopy()//保存屏幕图像到剪贴板
02  {
03      CDC* pDC = GetDC();                        //获取设备上下文
04      int cx = GetSystemMetrics(SM_CXSCREEN);//获取屏幕分辨率
05      int cy = GetSystemMetrics(SM_CYSCREEN);
06      HDC hScrDC = CreateDC("DISPLAY", NULL, NULL, NULL);
07      //创建屏幕上下文句柄
08      HBITMAP hBitmap = CreateCompatibleBitmap(hScrDC, cx, cy);
09      //创建位图对象
10      HGLOBAL hGlobal =(HGLOBAL)(UINT)hBitmap;//申请位图全局句柄
11      OpenClipboard();                           //打开剪贴板
12      SetClipboardData(MF_BITMAP, (HANDLE)hGlobal);//设置剪贴板数据
13      CloseClipboard();                          //关闭剪贴板
14  }
```

上面的代码首先调用 GetSystemMetrics()函数获取当前屏幕的大小范围。然后调用 CreateDC()函数创建屏幕 DISPLAY 画布，并使用此画布创建 HBITMAP 句柄。最后将获得的位图句柄通过 SetClipboardData()函数保存到剪贴板中。

25.6.8　获取图像 RGB 值

要获取图像 RGB 值，可以使用 Bitmap 对象的 GetPixel()函数获取指定像素点的 RGB

值。指定位置通过点所在的横坐标的像素值和纵坐标的像素值确定。代码如下：

```
01  void CGDIControlView::OnMenuitemGetrgb()    //获取图像 RGB 值示例
02  {
03      Graphics graphics(m_hWnd);              //定义 Graphics 变量
04      Bitmap image(L"baby.JPG");              //创建 Image 对象
05      UINT width = image.GetWidth();
06      UINT height = image.GetHeight();        //获取图像尺寸
07      graphics.DrawImage(&image, 0, 30, width, height);//绘制图像
08      Color color;
09      image.GetPixel(width/2, height/2, &color);//获取中心点的像素颜色值
10      CString log;
11      log.Format("图像中心点的颜色值为：Alpha=%d;Red=%d;Green=%d;Blue=%d",
12          color.GetAlpha(),
13         color.GetRed(), color.GetGreen(), color.GetBlue());
14      //格式化颜色信息
15      CDC* pDC = GetDC();                     //获取设备上下文
16      pDC->TextOut(0, 0, log);                //输出颜色值
17  }
```

上面代码首先创建 Graphics 对象和 Bitmap 对象，并对其进行初始化。然后使用 DrawImage()函数显示图片，并通过调用 GetPixel()函数获得图像中间位置点的 RGB 颜色值。最后，将获取的颜色值的信息在屏幕上端显示出来。程序运行效果如图 25-38 所示。



图 25-38　获取图像 RGB 值运行效果

25.6.9　渐隐渐显的图像

使用透明度不同的实画刷与图像画刷组合，可以实现图像的渐隐。即定时减少实画刷的透明度，则图像就会慢慢地消失，实现图像渐隐的效果。而对于渐现图像，则通过设定图像上相应点像素的透明度越来越大进行实现。由于此效果需要定时处理，为了不影响主程序的运行，因此需要创建线程，在该线程中执行。代码如下：

```
01  void CGDIControlView::OnMenuitemShimage()//渐隐渐显的图像处理函数
02  {
```

```
03      DWORD dwThreadId;                       //定义线程 ID
04      hThread = CreateThread(NULL,0, (LPTHREAD_START_ROUTINE)
05          ThreadShimageFunc,
06          (LPVOID)m_hWnd, 0, &dwThreadId);//创建绘制渐隐渐现图像的线程
07  }
```

上面代码显示了当用户要渐隐渐显图像时，程序会启动一个线程。线程运行代码如下：

```
01  DWORD WINAPI ThreadShimageFunc( LPVOID lpParam )//线程处理函数
02  {
03      Graphics graphics((HWND)lpParam);             //定义 Graphics 变量
04      graphics.SetSmoothingMode(SmoothingModeAntiAlias);
05      Bitmap image(L"baby.JPG");                    //创建位图对象
06      UINT width = image.GetWidth();
07      UINT height = image.GetHeight();              //获取图像尺寸
08      TextureBrush brush(&image);                   //绘制图像画刷
09      graphics.FillRectangle(&brush, Rect(0,0,width,height));
10      //使用图像画刷填充矩形
11      for (int i = 0; i < 255; i++)                 //实现渐隐效果
12      {
13          SolidBrush solidBrush(Color(i,0xA9,0xA9, 0xA9));
14          //设置不同更透明度的画刷
15          graphics.FillRectangle(&solidBrush, Rect(0,0,width,height));
16          //绘制矩形
17          Sleep(100);                               //绘制延时
18      }
19      graphics.Clear(Color.White);                  //清除界面上的白色像素值
20      for (i = 0; i <255; i++)                      //实现渐现效果
21      {
22          Color color, colorTemp;
23          for(INT iRow = 0; iRow < (INT)height; iRow++)
24          {
25              for(INT iColumn = 0; iColumn < (INT)width; iColumn++)
26              {
27                  image.GetPixel(iColumn, iRow, &color);
28                  //获取对应点的图像中的颜色值
29                  colorTemp.SetValue(color.MakeARGB(i, color.GetRed(),
30                      color.GetGreen(), color.GetBlue()));//设置颜色值
31                  image.SetPixel(iColumn, iRow, colorTemp);
32                  //设置像素对应的颜色值
33              }
34          }
35          graphics.DrawImage(&image, 0, 0, width, height);//绘制图像
36          Sleep(100);                               //绘制延时
37      }
38      return 0;
39  }
```

上面代码首先创建 Graphics 对象和 Image 对象，并对其进行初始化。第一步实现图像渐隐的效果，先在画布上显示图片，然后通过实画刷的透明度越来越小控制图像渐渐消失。第二步实现图像渐显，通过设置点像素的透明度越来越大，控制图像渐显的效果。此程序的运行效果是动态的，在运行时可以看到，图像慢慢消失，一会儿又慢慢显示。程序运行效果如图 25-39 所示。

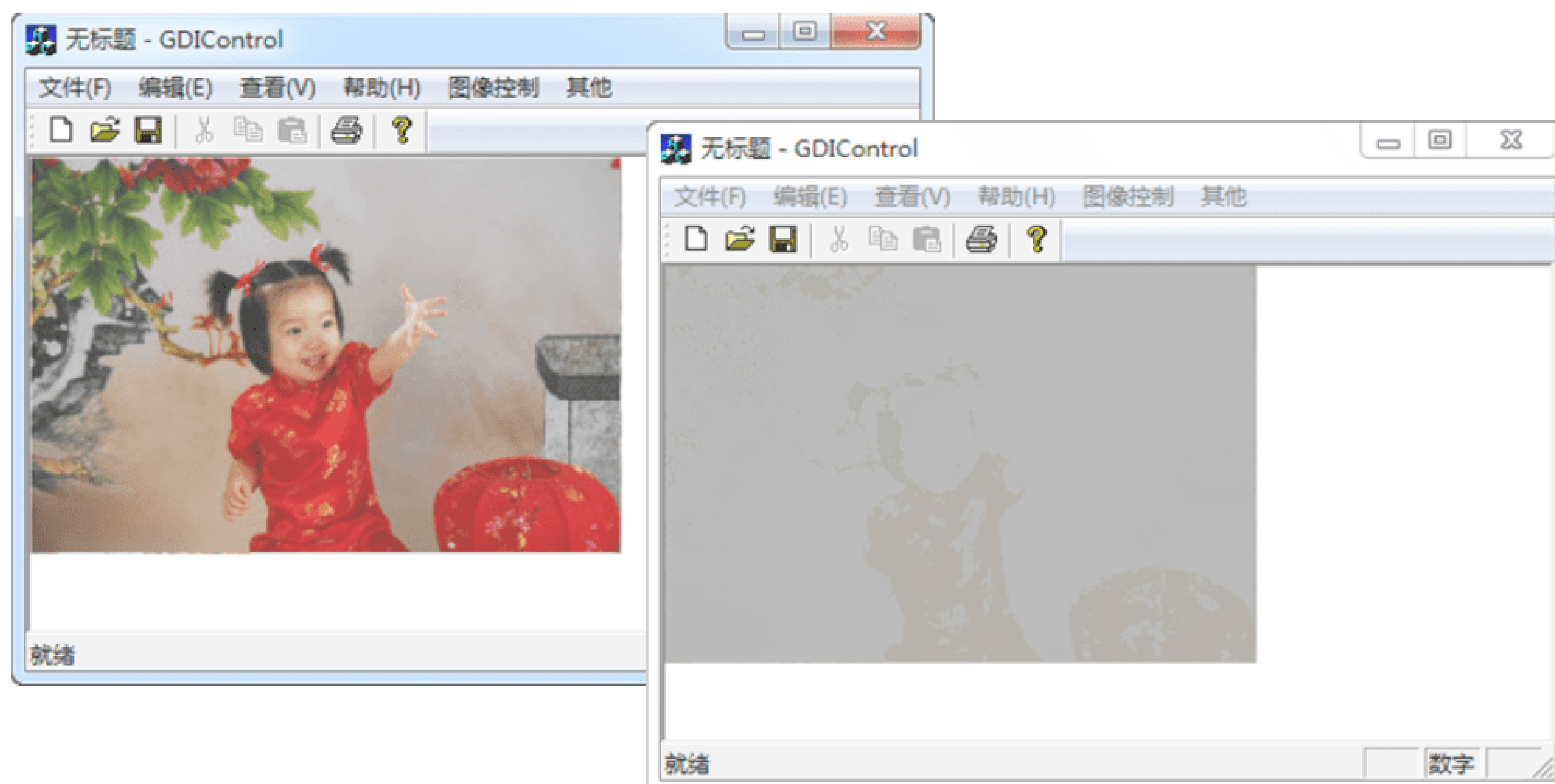


图 25-39　渐隐渐显的图像运行效果

25.6.10　保留椭圆中图片内容

使用 GDI+技术要保留椭圆中图片内容，分为以下两步。

（1）处理按钮事件。当用户触发按钮时，在屏幕上显示图片，并设置类变量 OperType 的值为 1，表示当前执行的操作是保留椭圆中图片内容。代码如下：

```
01  //保留椭圆中图片内容示例
02  void CGDIControlView::OnMenuitemSaveellipcontent()
03  {
04      Graphics graphics(m_hWnd);                       //定义 Graphics 变量
05      Image image(L"baby.JPG");                        //创建 Image 对象
06      UINT width = image.GetWidth();
07      UINT height = image.GetHeight();                 //获取图像尺寸
08      graphics.DrawImage(&image, 0, 0, width, height);//绘制原始图像
09      OperType = 1;
10  }
```

（2）处理鼠标抬起事件。使用当前点的坐标和鼠标按下事件的点的坐标进行计算，计算出椭圆形的区域，并使用 DrawPath()函数绘制区域路径，通过 SetClip()函数设置有效剪切区域为选择的椭圆形区域，最后重新绘制图片并将操作状态还原。代码如下：

```
01  //左键抬起的处理函数
02  void CGDIControlView::OnLButtonUp(UINT nFlags, CPoint point)
03  {
04      endPoint = point;
05      if (OperType == 1)
06      {                                                //保留椭圆中的内容
07          Graphics graphics(m_hWnd);                   //创建位图对象
08          graphics.Clear(Color.White);                 //清除其中的白色像素值
09          GraphicsPath path;
10          //增加椭圆区域
11          path.AddEllipse(beginPoint.x, beginPoint.y,
12              (endPoint.x-beginPoint.x), (endPoint.y-beginPoint.y));
```

```
13            Region region(&path);
14            Pen pen(Color(255, 0, 0, 255));          //定义画笔
15            graphics.DrawPath(&pen, &path);          //绘制此椭圆
16            graphics.SetClip(&region);               //设置剪切的区域
17            Image image(L"baby.JPG");                //装载图片
18            UINT width = image.GetWidth();
19            UINT height = image.GetHeight();         //获取图像尺寸
20            graphics.DrawImage(&image, 0, 0, width, height);//绘制图片
21            OperType = 0;
22        }
23        …
24    }
```

运行上面的代码，单击“保留椭圆中图片内容”按钮后，在屏幕上选择要保留的内容矩形框，则内接椭圆形范围内的内容会保存下来。程序运行效果如图 25-40 所示。

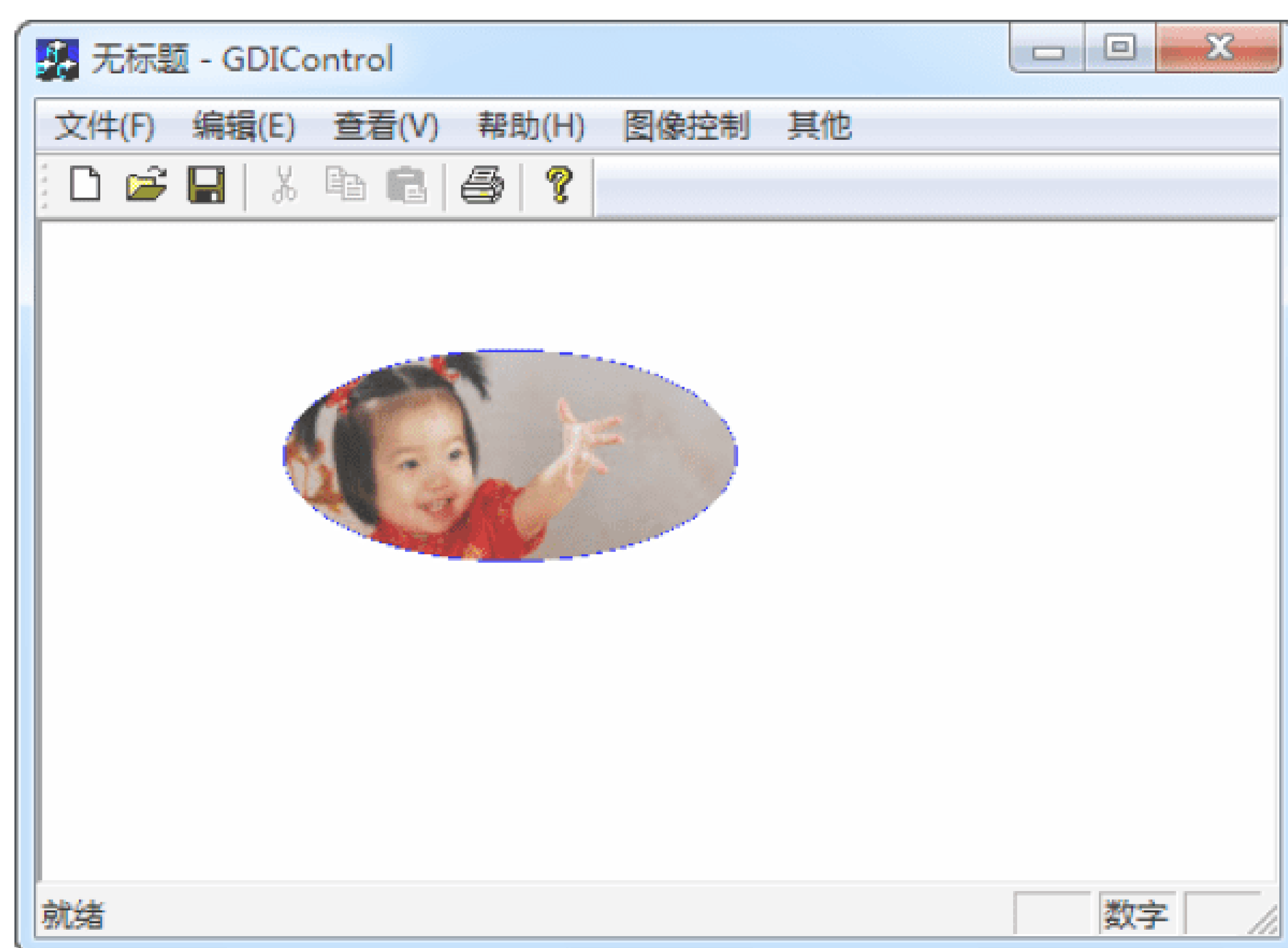


图 25-40　保留椭圆中图片内容运行效果

25.6.11　去除椭圆下的图片内容

使用 GDI+技术要去除椭圆下的图片内容，分为以下两步。

（1）处理按钮事件。当用户触发按钮时，在屏幕上显示图片，并设置类变量 OperType 的值为 2，表示当前执行的操作是去除椭圆下的图片内容。代码如下：

```
01  //去除椭圆下的图片内容示例
02  void CGDIControlView::OnMenuitemCutellipcontent()
03  {
04      Graphics graphics(m_hWnd);               //定义 Graphics 变量
05      Image image(L"baby.JPG");                //创建 Image 对象
06      UINT width = image.GetWidth();
07      UINT height = image.GetHeight();         //获取图像尺寸
08      graphics.DrawImage(&image, 0, 0, width, height);//绘制原始图像
09      OperType = 2;
10  }
```

（2）处理鼠标抬起事件。使用当前点的坐标和鼠标按下事件的点的坐标进行计算，计算出椭圆形的区域，调用 Region 对象的 Exclude()函数，反选选择的椭圆形范围，并使用

DrawPath()函数绘制区域路径，通过 SetClip()函数设置有效剪切区域为非选择的椭圆形区域，最后重新绘制图片并将操作状态还原。代码如下：

```
01  //左键按下的处理函数
02  void CGDIControlView::OnLButtonUp(UINT nFlags, CPoint point)
03  {
04      endPoint = point;
05      ……
06      else if (OperType == 2)                     //去除椭圆中的内容
07      {
08          Graphics graphics(m_hWnd);              //创建位图对象
09          graphics.Clear(Color.White);            //清除其中的白色像素值
10          CRect rect;
11          GetClientRect(&rect);                   //获取工作区域
12          Region region1(Rect(rect.left, rect.top, rect.right, rect.
13              bottom));
14          GraphicsPath path;
15          //记录椭圆路径信息
16          path.AddEllipse(beginPoint.x, beginPoint.y,
17              (endPoint.x-beginPoint.x), (endPoint.y-beginPoint.y));
18          Region region2(&path);                  //定义椭圆外区域
19          region1.Exclude(&region2);
20          Pen pen(Color(255, 0, 0, 255));         //创建画笔
21          graphics.DrawPath(&pen, &path);         //绘制椭圆
22          graphics.SetClip(&region1);             //设置剪切区域
23          Image image(L"baby.JPG");               //装载 Image
24          UINT width = image.GetWidth();
25          UINT height = image.GetHeight();        //获取图像尺寸
26          graphics.DrawImage(&image, 0, 0, width, height);
27          OperType = 0;
28      }
29      …
30  }
```

运行上面的代码，单击“去除椭圆下的图片内容”按钮后，在屏幕上选择要去除内容的椭圆形所在的矩形框，则其内接椭圆形范围内的内容会被去除。程序运行效果如图 25-41 所示。

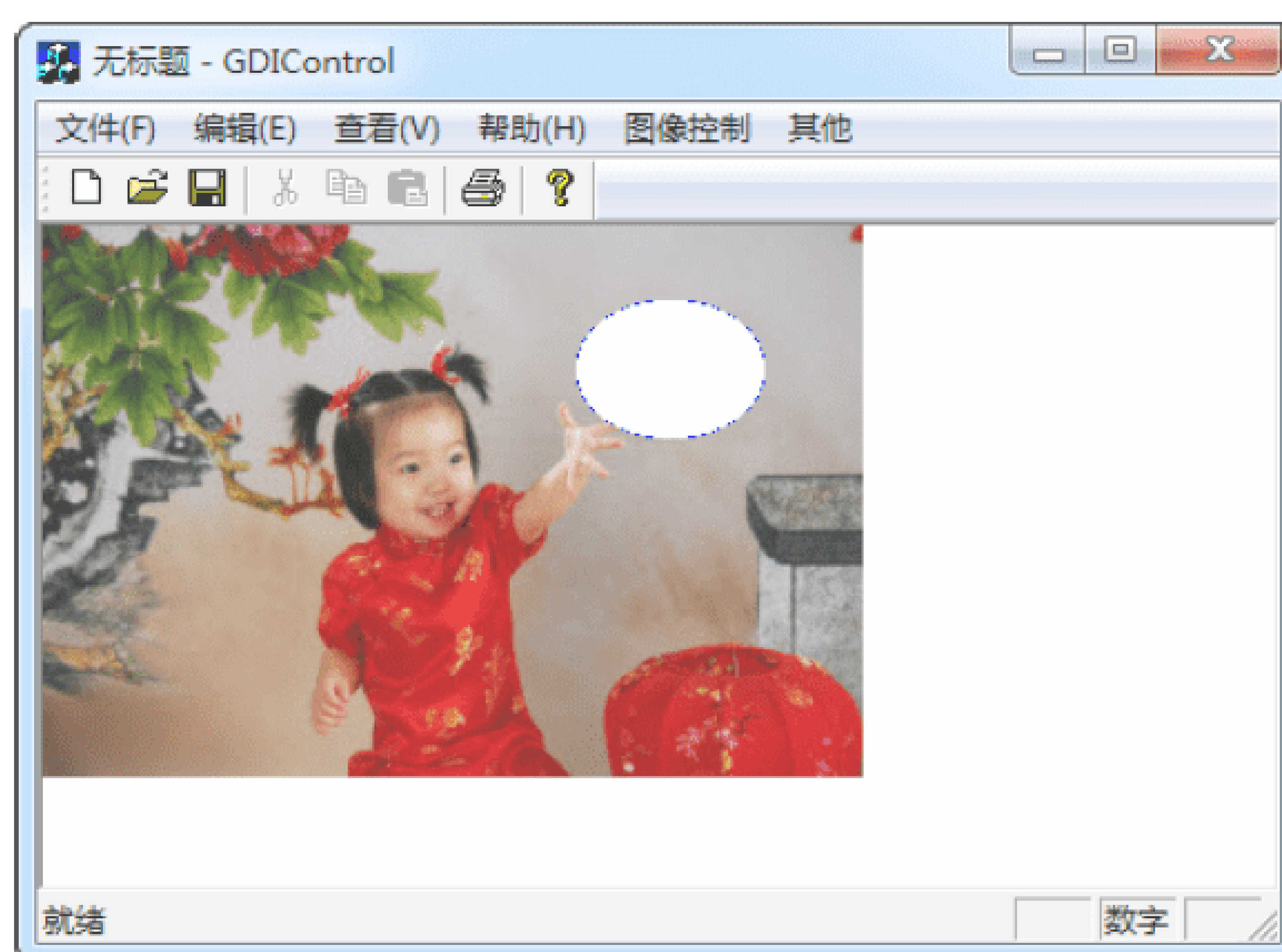


图 25-41　去除椭圆下的图片内容运行效果

25.7　本 章 小 结

本章主要讲述了如何通过 GDI 和 GDI+进行图形和图像方面的编程。本章重点介绍了画笔和画刷的使用、图像基础技术、一些特殊曲线和图像特效及控制的实现。本章难点是图像编程的流程，以及各种图像特效的算法。第 26 章将介绍 VC 中多媒体有关声音和动画的开发方法。

25.8　习　　题

1．创建基于单文档的项目，再任意装载一幅位图，添加新的菜单“我的菜单”，在新菜单下添加菜单项“绘制位图”。编程：当单击菜单项“绘制位图”时，在客户区显示位图，位图居中显示，显示的位图的长和宽分别是客户区长和宽的一半。

【思路】参考 25.1.2 小节所讲的示例，只是修改了位图绘制的大小。

2．在第 1 题的基础上，在菜单“我的菜单”下添加菜单项“绘制矩形”。编程：当单击菜单项“绘制矩形”时，在客户区的左上角显示一个红色的填充矩形，矩形的长和宽为客户区长和宽的四分之一。

【思路】与 25.2.4 小节所讲的示例相比，绘制的几何图形不同，画刷填充的颜色不同。

3．在第 1 题的基础上，添加新的菜单“处理图片”，并完成以下操作。

（1）为菜单“处理图片”添加菜单项“绘制网格”。单击此菜单项完成的功能是：在图片上添加蓝色的网格线并输出在客户区的左上角，使用图片资源本身的大小。

（2）鼠标左键单击图像上的任意位置时，在客户区的底部居中显示此图片的 RGB 值。

【思路】操作（1）和（2）可以分别参考 25.6.2 小节和 25.6.8 小节的示例来完成。

第 26 章　声音与动画编程

第 25 章讲述了图形与图像的处理，本章就讲述在 Visual Studio 2010 环境下，如何使用 Win32、GDI、GDI+和 DirectShow 等开发接口实现声音与动画的处理。一般情况下，声音处理也称为音频处理，动画、图形等处理也称为视频处理。Windows 提供对音频和视频操作的 Win32 API，同时提供了 Direct Show，使用其可以实现对音频和视频的捕捉。本章将讲述如何使用这些接口完成特定的音频和视频处理。

26.1　多媒体声音控制

Windows API 中提供了操作多媒体声音的函数。本节简单介绍声音录制与播放的实现，如何编写可以选择播放曲目的 CD 播放器，同时还介绍通过 Windows API 控制音量和左右声道以及利用 PC 的喇叭播放声音，还将介绍实现定时播放 WAV 文件以及具有记忆功能的 MP3 播放器，最后介绍使用 Visual Studio 2010 编写 MIDI 文件播放程序。

26.1.1　录制与播放声音

Windows API 中提供了可以简单地播放声音的函数 PlaySound()，它可以播放声音文件、声音资源或系统声音。其函数原型如下：

```
BOOL PlaySound(
  LPCSTR pszSound,                    //指定要播放的声音的名称
  HMODULE hmod,                       //包含载入资源的可执行文件的句柄
  DWORD fdwSound );                   //指定播放声音的选项
```

其中 fdwSound 参数指定播放声音的选项，可以是下列选项的组合。

- SND_APPLICATION：播放与应用程序相关的声音。
- SND_ALIAS：播放的声音是系统定义的声音。
- SND_ALIAS_ID：pszSound 参数是预定义的声音标识符。
- SND_ASYNC：声音播放是异步的，PlaySound()函数在开始播放声音后立即返回。要终止播放声音，则通过 PlaySound()函数传入 NULL 参数实现。
- SND_FILENAME：pszSound 参数指定要播放的声音文件名称。
- SND_LOOP：重复播放声音，必须与 SND_ASYNC 选项一起使用。
- SND_MEMORY：系统声音文件装载入内存。
- SND_NODEFAULT：不适用默认声音事件。如果要播放的声音没有查找到，则函数不播放任何声音，直接返回。

- SND_NOSTOP：如果正有其他程序在占用资源播放声音，则不会停止当前播放的声音，而直接返回。
- SND_NOWAIT：如果设备正忙，则函数立即返回。
- SND_PURGE：停止声音播放。
- SND_RESOURCE：播放的声音是资源，必须包含在 hmod 参数中的指定模块中。
- SND_SYNC：异步播放声音。

播放声音文件成功，返回 true；如果播放声音文件失败，则返回 false。下面代码显示了如何播放 Windows 启动声音文件。

```
01  void CSoundSampleDlg::OnButtonPlay()        //播放声音示例
02  {
03      if (!PlaySound("C:\\Windows XP 启动.wav", NULL,
04          SND_SYNC | SND_NODEFAULT))
05          WriteLog("播放声音文件出现错误");
06  }
```

使用 waveIn 系列的函数可以录制声音文件。代码如下：

```
01  //录制声音示例
02  void CSoundSampleDlg::OnButtonRecord()
03  {
04      MMRESULT mmResult;                          //定义操作结果变量
05      WAVEFORMATEX m_waveformat;                  //定义音频格式
06      m_waveformat.wFormatTag=WAVE_FORMAT_PCM;//赋值为 PCM 格式
07      m_waveformat.nChannels=1;                   //赋值通道为 1
08      m_waveformat.nSamplesPerSec=11025;          //设置每秒采样 11025
09      m_waveformat.nAvgBytesPerSec=11025;         //设置每秒平均字节为 11025
10      m_waveformat.nBlockAlign=1;                 //设置对齐方式
11      m_waveformat.wBitsPerSample=8;          //设置每个采样标本中的 Bit 位数
12      m_waveformat.cbSize=0;                      //设置结构长度
13      //播放音频
14      if ((mmResult = waveInOpen(&m_hWaveIn, 0, &m_waveformat,
15          (DWORD)this->m_hWnd, NULL,
16          CALLBACK_WINDOW))!= MMSYSERR_NOERROR)
17      {
18          if (mmResult == MMSYSERR_ALLOCATED)
19              WriteLog("音频设备已经被其他设备占用");
20          else if(mmResult == MMSYSERR_BADDEVICEID)
21              WriteLog("指定的音频设备标识符超出范围");
22          else if (mmResult == MMSYSERR_NODRIVER)
23              WriteLog("没有准备好音频设备");
24          else if (mmResult == MMSYSERR_NOMEM)
25              WriteLog("没有分配好内存");
26          else if (mmResult == WAVERR_BADFORMAT)
27              WriteLog("错误的音频格式");
28          else
29              WriteLog("打开音频输入设备失败");
30          return;
31      }
32      const int MAX_WAVEDATA_LENGTH = 90040;  //定义音频数据的最大长度
33      inbuf = new char[MAX_WAVEDATA_LENGTH];  //定义音频数据缓冲区
34      m_wavehdr.lpData=inbuf;                     //赋值音频句柄数据缓冲区
35      m_wavehdr.dwBufferLength=MAX_WAVEDATA_LENGTH;//赋值缓冲区长度
```

```
36      m_wavehdr.dwBytesRecorded=0;                //赋值字节记录
37      m_wavehdr.dwUser=0;
38      m_wavehdr.dwFlags=0;
39      m_wavehdr.dwLoops=0;
40      m_wavehdr.lpNext=NULL;
41      m_wavehdr.reserved=0;
42      //准备录音
43      if (waveInPrepareHeader(m_hWaveIn, &m_wavehdr,
44          sizeof(m_wavehdr))!= MMSYSERR_NOERROR)
45      {
46          WriteLog("为录音设备准备缓存函数失败");
47          return;
48      }
49      //为录音设备增加输入缓冲区
50      if (waveInAddBuffer(m_hWaveIn, &m_wavehdr, sizeof(m_wavehdr))
51          != MMSYSERR_NOERROR)
52      {
53          WriteLog("给输入设备增加一个缓存失败");
54          return;
55      }
56      if (waveInStart(m_hWaveIn)!= MMSYSERR_NOERROR)
57      {
58          WriteLog("开始录音失败");
59          return;
60      }
61      WriteLog("开始录音……");
62  }
```

上面代码中，首先调用 waveInOpen()函数打开音频输入设备，然后调用 waveInPrepareHeader()函数为录音设备准备缓存空间，调用 waveInAddBuffer()函数为录音设备增加缓冲区，最后调用 waveInStart()函数启动录音。录音完毕，可以调用 waveIn 系列函数停止录音。代码如下：

```
01  void CSoundSampleDlg::OnButtonStoprecord()          //停止录音代码
02  {
03      if (waveInReset(m_hWaveIn)!= MMSYSERR_NOERROR)  //停止录音
04      {
05          WriteLog("停止录音失败");
06          return;
07      }
08      MMRESULT mmResult;                              //定义操作结果
09      if ((mmResult = waveInUnprepareHeader(m_hWaveIn,
10          &m_wavehdr, sizeof(m_wavehdr)))
11          != MMSYSERR_NOERROR)                        //释放输入设备
12      {
13          if (mmResult == MMSYSERR_INVALHANDLE)
14              WriteLog("设备句柄无效");
15          else if (mmResult == MMSYSERR_NODRIVER)
16              WriteLog("没有准备好设备驱动");
17          else if (mmResult == MMSYSERR_NOMEM)
18              WriteLog("没有分配或锁定内存");
19          else if (mmResult == WAVERR_STILLPLAYING)
20              WriteLog("正在播放声音");
21          else
22              WriteLog("清除缓存失败");
23          return;
24      }
```

```
25      if (waveInClose(m_hWaveIn)!= MMSYSERR_NOERROR)       //关闭录音
26      {
27          WriteLog("关闭录音设备失败");
28          return;
29      }
30      WriteLog("录制声音完成");
31  }
```

上面代码中，首先调用 waveInReset()函数停止录音，然后调用 waveInUnprepareHeader()函数清除缓存，最后调用 waveInClose()函数关闭录音设备。可以根据需要在调用 waveInUnprepareHeader()函数前，保存录制好的声音。

26.1.2　可以选择曲目的 CD 播放器

Windows API 中提供了一组 MCI 函数，即媒体控制接口。提供了播放多媒体设备和录制多媒体资源文件的标准命令。这些命令为多种多媒体设备提供统一的访问接口。mciGetDeviceID()函数用于获取指定名称的设备标识符。其函数原型为：

```
MCIDEVICEID mciGetDeviceID( LPCTSTR lpszDevice );
```

参数 lpszDevice 用于指定设备名称或已知的设备别名。函数会返回与打开的设备相连的设备标识符。如果返回值为 0，表示获取关联的设备 ID 发生错误。返回的设备 ID，可以用在 mciSendCommand()函数中，对设备进行操作。mciSendCommand()函数原型为：

```
MCIERROR mciSendCommand(
  MCIDEVICEID IDDevice,        //指定命令消息的 MCI 设备标识符
  UINT uMsg,                   //指定命令消息，可以执行播放、暂停和快进等多种操作
  DWORD fdwCommand,            //表示命令消息的选项
  DWORD dwParam );             //包含命令消息的参数结构地址
```

调用 mciSendCommand()函数可以向指定的 MCI 设备发送命令消息。如果操作成功，返回 0，否则返回错误代码。使用 mciGetErrorString()函数可以获取错误代码所代表的原因。下面代码演示了使用 MCI 函数如何播放指定曲目的 CD。

```
01  void CSoundSampleDlg::OnButtonPlaycd()                //播放 CD 曲目
02  {
03      UpdateData(true);                                 //获取数据
04      MCIDEVICEID deviceID = mciGetDeviceID("G:\\");    //获取光驱 G 盘
05      DWORD dwPlay = m_Volumn;                          //获取音量
06      DWORD dwFlags;
07      MCI_DGV_PLAY_PARMS mciPlay;                       //定义播放参数
08      mciPlay.dwCallback = MAKELONG(m_hWnd,0);          //设置回放窗体句柄
09      mciPlay.dwFrom = mciPlay.dwTo = dwPlay;
10      dwFlags = MCI_NOTIFY;                             //设置通知消息
11      dwFlags |= MCI_DGV_PLAY_REVERSE;                  //设置播放消息
12      MCIERROR mciError;
13      //发送播放消息
14      if ((mciError=mciSendCommand(deviceID, MCI_PLAY,
15          dwFlags, (DWORD)(LPMCI_DGV_PLAY_PARMS)&mciPlay)) == 0)
16          WriteLog("正在播放曲目……");
17      else                                          //如果失败，则输出错误提示
18      {
```

```
19          char szErrorText[500]={0};
20          mciGetErrorString(mciError,szErrorText,sizeof(szErrorText));
21          WriteLog(szErrorText);
22      }
23  }
```

上面代码，使用 mciGetDeviceID()函数获取 CD 播放器的设备 ID，然后通过 mciSendCommand()函数播放编辑框中用户输入的曲目序号中的 CD 曲目。其中，第二个参数 MCI_PLAY 表示播放曲目，而 MCI_DGV_PLAY_PARMS 结构用于指定播放参数，dwFrom 和 dwTo 表示要播放曲目的开始序号和结束序号。如果在播放时发生错误，则调用 mciGetErrorString()函数返回失败原因，并在日志编辑框中显示出来。

26.1.3　控制音量

Windows API 中提供了可以控制音量的接口函数。使用 auxGetNumDevs()函数可以获取当前系统中安装的声卡数目。其函数原型为：

```
UINT auxGetNumDevs(VOID);
```

此函数的返回值为当前系统中安装的声卡设备数目。如果返回值为 0，表示当前系统中没有声卡或有错误发生。auxGetVolume()函数返回指定的音频输出设备的当前音量。其函数原型为：

```
MMRESULT auxGetVolume(
  UINT uDeviceID,                 //指定要查询当前音量的音频设备的标识符
  LPDWORD lpdwVolume);            //用于存放返回的音频设备的当前音量值
```

函数如果返回 0xFFFF，表示最大音量，返回 0x0000 表示静音。如果函数调用成功，则返回 MMSYSERR_NOERROR。auxSetVolume()函数设置指定音频输出设备的音量。其函数原型为：

```
MMRESULT auxSetVolume(
  UINT uDeviceID,                 //指定要设置音量的音频设备的标识符
  DWORD dwVolume);                //指定要设置的音频设备的音量
```

如果设置音量成功，则返回 MMSYSERR_NOERROR。下面的代码结合这 3 个函数，设置当前音量。

```
01  void CSoundSampleDlg::OnButtonCtrlvolumn() //设置音量
02  {
03      MMRESULT mmResult;                     //定义操作结果
04      DWORD dwDevNum;                        //定义设备数目变量
05      dwDevNum = ::auxGetNumDevs();          //获取设备数目
06      if (dwDevNum>0)                        //如果设备数目大于 1
07      {
08          DWORD dwVolume1;
09          mmResult=auxGetVolume(AUX_MAPPER, &dwVolume1);//获取设备音量
10          if (mmResult != MMSYSERR_NOERROR)  //判断操作结果
11          {
12              if (mmResult == MMSYSERR_BADDEVICEID)
13                  WriteLog("音频设备无效");
14              else
```

```
15                  WriteLog("获取当前音频设备的音量失败");
16              return;
17          }
18          UpdateData(false);
19          CString log;
20          DWORD dwVolume2 = m_Volumn;
21          mmResult=auxSetVolume(AUX_MAPPER, dwVolume2);//设置音量
22          if (mmResult != MMSYSERR_NOERROR)
23              log.Format("设置音量失败。原来音量=%d", dwVolume1);
24          else
25              log.Format("设置音量成功。原来音量=%d;设置后的音量=%d",
26              dwVolume1,dwVolume2);
27          WriteLog(log);
28      }
29      else
30          WriteLog("没有有效的音频设备");                   //输出错误提示
31  }
```

上面的函数，首先调用 auxGetNumDevs()函数获取当前安装的声卡的数目。如果当前系统中安装了声卡，则调用 auxGetVolume()函数获取声卡的当前音量，然后调用 auxSetVolume()函数设置用户指定的音量。

26.1.4　利用 PC 喇叭播放声音

使用 Beep()函数可以通过 PC 喇叭播放声音，此函数是同步的，直到播放完声音后，才会返回。其函数原型为：

```
BOOL Beep(
  DWORD dwFreq,           //指定播放的声音的频率，单位是赫兹，从 0x25～0x7FFF
  DWORD dwDuration);      //指定声音持续的时间，单位是毫秒
```

如果函数成功，返回非 0 值；如果失败，则返回 0，使用 GetLastError()函数可以获取错误原因。以下代码播放频率为 1570Hz 的声音，持续 5s。

```
::Beep(1570,5000);
```

26.1.5　定时播放 WAV 文件

使用 PlaySound()函数和定时器机制可以实现定时播放 WAV 文件。代码如下：

```
01  void CSoundSampleDlg::OnButtonPlaywav()         //播放 WAV 文件函数
02  {
03      SetTimer(200, 10000, NULL);                 //启动定时器处理函数
04  }
05  void CSoundSampleDlg::OnTimer(UINT nIDEvent)    //定时器处理函数
06  {
07      if (nIDEvent == 200)
08      {
09          CString sFileName = _T("Windows XP 启动.wav");
10          ::PlaySound(sFileName, NULL, SND_FILENAME); //播放声音文件
11      }
12      CDialog::OnTimer(nIDEvent);           //调用定时器类的基础定时函数
13  }
```

上面代码中，SetTimer()函数启动定时器，在定时器处理函数中，调用 PlaySound()函数播放指定的 WAV 文件。本例中，播放 Windows 系统中自带的启动声音。

26.1.6　播放 MIDI 文件

MIDI（Musical Instrument Digital Interface）即音乐器具数字接口。是一个电子键盘标准，定义了传输和存储音乐信息的协议。Windows 提供了一组接口库可以播放 MIDI 文件。MCIWnd 是控制多媒体设备的对话框类。接口库中包含与 MCIWnd 相连的函数、消息和宏，来提供增加多媒体播放或录制功能的简单方法，这些接口存放在 vfw32.lib 库中。其中使用 MCIWndCreate()函数可以打开 MCI 控制对话框，打开 MCI 设备或文件，其函数原型为：

```
HWND MCIWndCreate(
  HWND hwndParent,          //指定 MCI 对话框的父对话框
  HINSTANCE hInstance,      //指定与 MCIWnd 对话框相连的模块实例句柄
  DWORD dwStyle,            //指定定义对话框样式的选项
  LPSTR szFile);            //指定要打开的 MCI 设备或数据文件
```

如果打开成功，则返回 MCI 对话框的句柄，否则返回 0。

MCIWndPlay 宏发送命令到 MCI 设备，从当前位置开始播放文件。其函数原型为：

```
LONG MCIWndPlay( hwnd );    //指定 MCIWnd 对话框的句柄
```

如果函数成功，返回 0，否则返回错误代码。以下代码显示了如何在 VC 中调用 MIDI 文件播放程序。

```
01  void CSoundSampleDlg::OnButtonPlaymidi()         //播放处理函数
02  {
03      HWND hMCIWnd = MCIWndCreate(NULL, NULL, 0, "town.mid");
04      MCIWndPlay(hMCIWnd);                         //启动 MIDI 播放器
05  }
```

上面代码会启动 MIDI 播放器，并播放 town.mid 文件。

26.1.7　开发具有记忆功能的 MP3 播放器

通过 MCIWndGetPosition()宏接收当前 MCI 设备的播放位置，可以实现具有记忆功能的 MP3 播放器。宏原型为：

```
LONG MCIWndGetPosition( hwnd );     //指定 MCIWnd 对话框的句柄
```

此函数的返回值为当前位置值。MCIWndGetLength()函数可以返回正在播放的文件的长度。MCIWndStop()函数可以停止播放当前文件。MCIWndDestroy()函数可以关闭指定的 MCIWnd 对话框。下面的代码使用这些函数，实现一个具有记忆功能的 MP3 播放器。

```
01  void CSoundSampleDlg::OnButtonPlaymp3()     //具有记忆功能的 MP3 播放器
02  {
03      int type=0;
04      if (m_hWndMCI == NULL)                  //判断 MCI 句柄是否为 NULL
05      {
```

```
06          fileName="雨中节奏.mp3";                //赋值 mp3 文件名
07          //创建 MIDI 播放器
08          m_hWndMCI=MCIWndCreate(m_hWnd, AfxGetInstanceHandle(),
09              type,fileName);
10          MCIWndPlay(m_hWndMCI);                  //启用 MIDI 播放器
11          return;
12      }
13      //为 MIDI 播放器设置新文件名
14      m_hWndMCI=MCIWndCreate(m_hWnd,AfxGetInstanceHandle(),
15          type, fileName);
16      long nLength = MCIWndGetLength(m_hWndMCI);      //获取文件长度
17      MCIWndPlayFromTo(m_hWndMCI, m_lPosition, nLength);//定位播放
18  }
19  void CSoundSampleDlg::OnButtonStopmp3()             //停止播放 MP3
20  {
21      //停止播放 MP3
22      //获取文件名
23      MCIWndGetFileName(m_hWndMCI, fileName.GetBuffer(1000), 1000);
24      m_lPosition = MCIWndGetPosition(m_hWndMCI);//获取当前播放的位置
25      MCIWndStop(m_hWndMCI);                          //停止播放
26      MCIWndDestroy(m_hWndMCI);                       //销毁 MIDI 播放器
27  }
```

上面代码中 OnButtonPlaymp3()函数实现播放 MP3 文件的功能。其中会首先判断是否打开过播放器，如果是第一次打开播放器，则会初始化播放的文件为“雨中节奏.mp3”，并从头开始播放；如果不是第一次打开播放器，则会打开上次关闭时播放的文件，并从上次关闭时播放到的位置开始播放 MP3 文件。OnButtonStopmp3()函数是停止播放按钮的处理函数。关闭 MCIWnd 对话框之前，记录正在播放的 MP3 文件和当前的播放位置，并停止播放 MP3 文件。这样，在播放 MP3 文件后，再次打开时，会从上次停止的位置开始播放 MP3 文件。

26.2　多媒体应用

使用多媒体可以增加程序功能，实现多种效果。本节将介绍几个有关多媒体的应用。26.2.1 小节和 26.2.2 小节介绍屏幕保护程序的创建，包括滚动字体的屏幕保护程序和相册屏幕保护程序的编写。26.2.3 小节介绍如何编写画图程序。通过本节的学习，读者应该能根据自己的需求编写适合的应用程序。

26.2.1　滚动字体作屏保

通过设置程序的窗体的样式和定时器机制以及 GDI 的图形绘制功能，可以实现屏幕保护的功能。本小节介绍如何实现滚动字体的屏幕保护程序。步骤如下。

（1）创建实现屏幕保护功能的对话框类 CPhotoScreenWnd，此对话框类在 Create()函数中创建全屏显示的黑色背景对话框，并且启动定时器。代码如下：

```
01  BOOL CPhotoScreenWnd::Create() //创建图片屏保
02  {
03      if (lpszClassName==NULL)     //判断类名是否为 NULL，如果为 NULL，则注册类
```

```
04      {
05          lpszClassName=AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW,
06              ::LoadCursor(AfxGetResourceHandle(),
07              MAKEINTRESOURCE(IDC_NOCURSOR)));
08      }
09      //获取屏幕分辨率
10      CRect rect(0,0,::GetSystemMetrics(SM_CXSCREEN),
11          ::GetSystemMetrics(SM_CYSCREEN));
12      //创建顶层窗口
13      CreateEx(WS_EX_TOPMOST,lpszClassName, T(""),WS_VISIBLE|WS_POPUP,
14          rect.left,rect.top,rect.right-rect.left,rect.bottom-rect.top,
15          GetSafeHwnd(),NULL,NULL);
16      SetTimer(m_idTimer, 500, NULL);//开启定时器
17      return true;
18  }
```

上面的函数会启动定时器，定时器的处理函数会在屏幕上显示文字，并增加文字的位置偏移量。在屏幕上显示文字的函数代码如下：

```
01  void CPhotoScreenWnd::DrawText(CDC& dc, int nIndex)//绘制文本函数
02  {
03      RedrawWindow();                                  //重绘窗体
04      int width = ::GetSystemMetrics(SM_CXSCREEN);
05      int height = ::GetSystemMetrics(SM_CYSCREEN);    //获取屏幕分辨率
06      int colWidth = width/m_nTextCount;               //设置列宽
07      //在指定位置输出文本内容
08      dc.TextOut(nIndex*colWidth, height/2, "欢迎进入屏保测试");
09  }
```

上面的函数会根据当前显示的屏幕来确定本次显示文字的横坐标位置，实现文字的水平滚动。如果要实现文字的垂直滚动，则需要根据当前显示的屏幕来确定本次显示文字的纵坐标位置。屏幕保护程序在接收到用户输入时退出屏幕保护程序。因此，需要处理键盘按键和鼠标单击事件，当接收到这些输入时，会退出屏保程序。代码如下：

```
01  void CPhotoScreenWnd::OnKeyDown(UINT nChar,
02                                  UINT nRepCnt,UINT nFlags)
03  {
04      PostMessage(WM_CLOSE);                      //鼠标按下时，退出屏保程序
05  }
```

上面代码只显示了当按下键盘键时，会发送 WM_CLOSE 消息，退出屏幕保护程序，其他输入事件的处理是相同的。

（2）在应用程序类 CPhotoScreenSaverApp 的 InitInstance()函数中，调用屏幕保护对话框，代码如下：

```
01  BOOL CPhotoScreenSaverApp::InitInstance()        //应用程序实例化函数
02  {
03      …
04      CPhotoScreenWnd* pWnd=new CPhotoScreenWnd;  //创建屏保窗体句柄
05      pWnd->Create();                             //创建窗体
06      m_pMainWnd=pWnd;                            //记录句柄
07      …
08  }
```

上面代码会创建屏幕保护对话框类 CPhotoScreenWnd，并调用其 Create()函数，显示屏

保对话框。

（3）编译、链接并生成应用程序，将生成的屏幕保护程序的 EXE 文件的扩展名修改为 SCR，并将其放置在 Windows 系统目录下。这样在设置屏保的对话框中就可以预览自定义的滚动字体的屏幕保护程序，如图 26-1 所示。



图 26-1　设置屏幕保护程序

26.2.2　相册作屏保

相册屏幕保护程序的创建与滚动字体的屏幕保护程序的创建方法是一样的，都是通过定时器定时更新屏幕内容来实现的，其区别仅在于，相册屏幕保护程序的定时器会执行 DrawBitmap()函数，每次绘制一幅位图。用户可以根据需要自己设置位图的个数，本小节中仅使用两幅位图实现滚动播放相片的功能。代码如下：

```
01  void CPhotoScreenWnd::DrawBitmap(CDC& dc, int nIndex)//绘制位图
02  {
03      CDC dcMem;                                      //定义设备上下文
04      dcMem.CreateCompatibleDC(&dc);                  //创建内存上下文
05      CBitmap m_Bitmap;                               //定义位图对象
06      m_Bitmap.LoadBitmap(IDB_BITMAP1+nIndex);        //装载位图
07      dcMem.SelectObject(m_Bitmap);                   //选择位图
08      dc.BitBlt(0,0, 1276, 854,&dcMem,0,0,SRCCOPY);   //绘制位图
09  }
```

上面代码是定时器的定时执行函数，其功能是以全屏的方式显示位图，并为位图计数分配新值，以实现动态相册播放的屏幕保护效果。按照 26.2.1 小节中介绍的方法，生成并

设置屏幕保护程序，即可实现动态播放相片的屏幕保护程序。

26.2.3　设计画图程序

除了通过程序控制图形的绘制外，经常会遇到需要根据用户的输入来绘制指定样式的图形，本小节以一个简单的例子讲解如何设计画图程序。介绍如何通过捕获用户的输入来绘制矩形和直线。读者可以根据自己的需要，增加其他形状和样式的绘制。设计画图程序主要分为两个步骤。

（1）处理按钮事件。此处定义了 3 个按钮，分别是绘制直线、绘制矩形和切换成箭头。在这 3 个按钮的处理函数中，分别将全局变量 type 的值设置为 1、2、0。代码如下：

```
01  void CDrawPictureSampleView::OnMenuitemDrawLine()
02  {
03      //绘制直线
04      type = 1;
05  }
06  void CDrawPictureSampleView::OnMenuitemDrawRect()
07  {
08      //绘制矩形
09      type = 2;
10  }
11  void CDrawPictureSampleView::OnMenuitemArrow()
12  {
13      //箭头
14      type = 0;
15  }
```

（2）处理鼠标按下事件和鼠标抬起事件。当鼠标按下时，会记录鼠标按下时的鼠标位置点，并存入全局变量 ptBegin 中。当鼠标抬起时，会根据当前的操作种类来执行相应的操作。代码如下：

```
01  //鼠标按下处理函数
02  void CDrawPictureSampleView::OnLButtonDown(UINT nFlags,
03      CPoint point)
04  {
05      ptBegin.x = point.x;
06      ptBegin.y = point.y;
07      CView::OnLButtonDown(nFlags, point);
08  }
09  void CDrawPictureSampleView::OnLButtonUp(UINT nFlags,
10      CPoint point)
11  {                                                   //鼠标抬起处理函数
12      CDC* pDC = GetDC();                             //获取设备上下文
13      if (type == 1)
14      {
15          hPen = CreatePen(PS_SOLID,8, RGB(255,0,255));//创建画笔
16          pDC->SelectObject(hPen);                    //选择画笔
17          ptEnd.x = point.x;
18          ptEnd.y = point.y;                          //记录点信息
19          pDC->MoveTo(ptBegin.x, ptBegin.y);          //移动到起点
20          pDC->LineTo(ptEnd.x, ptEnd.y);              //绘制直线
21          DeleteObject(hPen);                         //删除画笔
22          DeleteDC(pDC->m_hDC);                       //删除上下文
```

```
23      }
24      else if (type == 2)
25      {
26          hPen = CreatePen(PS_SOLID,5,RGB(0,255,0));  //创建画笔
27          hBrush = CreateSolidBrush(RGB(125, 125, 0));//创建画刷
28          pDC->SelectObject(hPen);                    //选择画笔
29          pDC->SelectObject(hBrush);                  //选择画刷
30          ptEnd.x = point.x;
31          ptEnd.y = point.y;
32          //绘制矩形
33          Rectangle(pDC->m_hDC,ptBegin.x, ptBegin.y,
34              ptEnd.x, ptEnd.y);
35          DeleteObject(hPen);                         //删除画笔
36          DeleteObject(hBrush);                       //删除画刷
37          DeleteDC(pDC->m_hDC);                       //删除上下文句柄
38      }
39      CView::OnLButtonUp(nFlags, point);
40  }
```

在上面代码中，OnLButtonDown()函数是鼠标按下事件的处理函数。OnLButtonUp()函数是鼠标抬起事件的处理函数，若当前命令是绘制直线，会创建宽度为 8 的粉红色画笔，在画布上绘制从鼠标按下点开始到鼠标抬起点结束的直线。若当前命令是绘制矩形，会绘制绿色边框、RGB（125，125，0）颜色为填充颜色的矩形，此矩形的对角线点的坐标是鼠标按下点和鼠标抬起点。读者可以根据需要，增加绘制多边形、三角形和折线等其他形式的图形工具。程序的运行效果如图 26-2 所示。

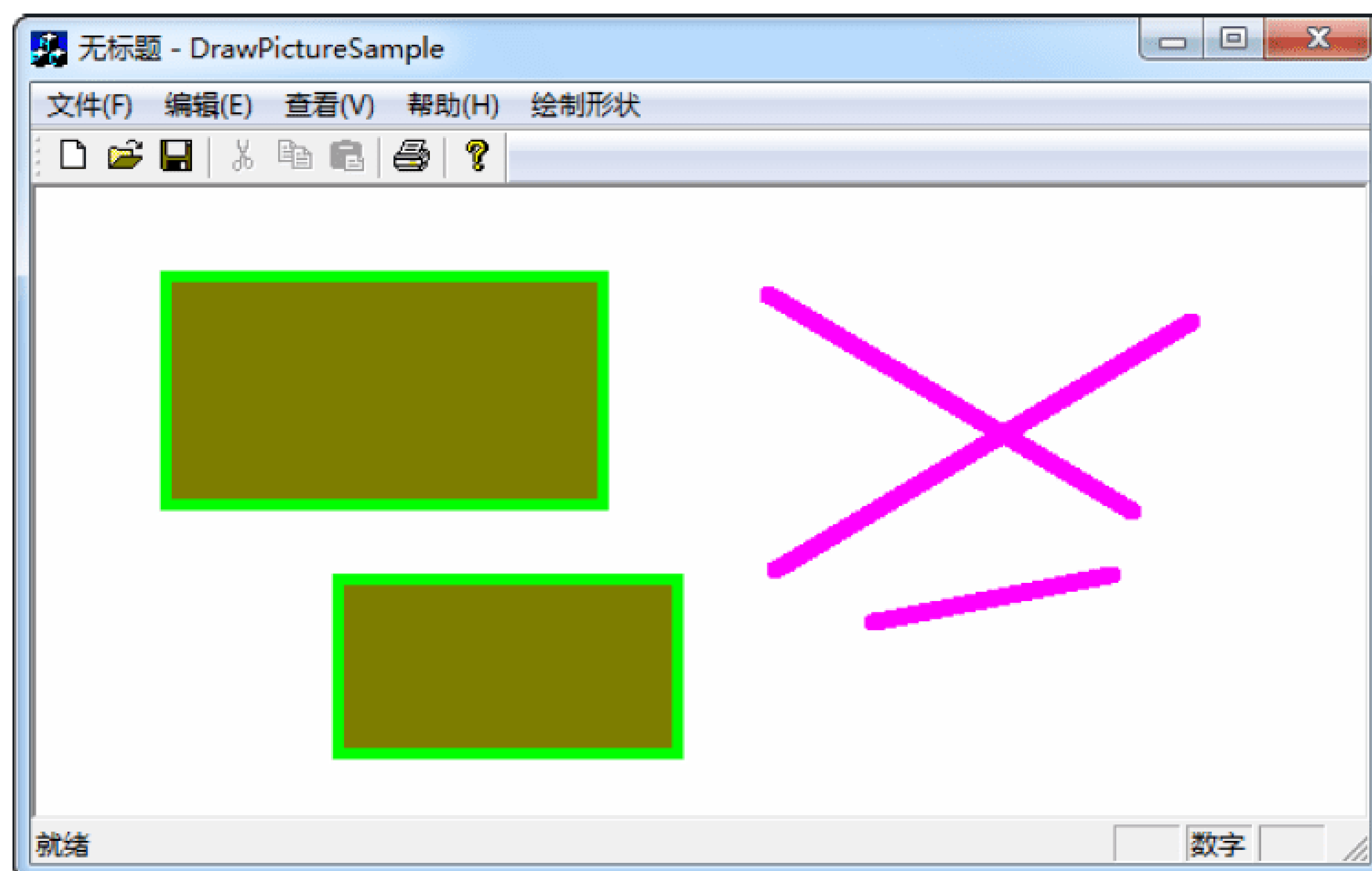


图 26-2　画图程序运行效果

26.3　动 画 效 果

除了实现静态的多媒体应用外，在 VC 中可以通过编程实现动画效果。本节将介绍部分动画效果的实现。26.3.1 小节介绍实现标题栏和任务栏的动画图标。26.3.2 小节介绍使用 DrawIcon()函数实现图标动画。26.3.3 小节介绍系统托盘中的动态图标的实现。

26.3.1　标题栏动画图标

通过定时器，可以实现标题栏显示动画图标的效果。在程序启动时，首先设置图标列表，并启动定时器，定时器每次获取当前索引下的图标句柄，并发送 WM_SETICON 消息给主对话框，这样，标题栏看上去显示的是动画图标。代码如下：

```
01  //设置大图标
02  BOOL CMainFrame::SetTBImageList(int imageListID,
03                                  int iMaxIcons, COLORREF tc)
04  {
05      if(iMaxIcons <= 0)
06          return false;                          //判断传入的参数有效性
07      m_iMaxTBIcon = iMaxIcons;                  //记录变量值
08      //创建图标列表
09      VERIFY(m_TBImgList.Create(imageListID, 16, 1, tc));
10      return true;                               //函数返回
11  }
12  BOOL CMainFrame::ShowTBNextIcon()              //设置下一个图标
13  {
14      if(m_TBImgList.m_hImageList == NULL)
15          return false; //判断图标列表是否有效
16      m_iTBIconCounter++;                        //图标计数器增 1
17      if(m_iTBIconCounter >= m_iMaxTBIcon)
18          //如果大于最大值，则归 0
19          m_iTBIconCounter =0;
20      hTBIcon = m_TBImgList.ExtractIcon(m_iTBIconCounter);//获取图标
21      //设置新图标，并保存原来的图标
22      HICON hPrevIcon =
23          (HICON) AfxGetMainWnd()->SendMessage(WM_SETICON,true,
24          (LPARAM)hTBIcon);
25      //释放原来的图标
26      if (hPrevIcon)
27          DestroyIcon(hPrevIcon);
28      return true;                               //函数成功返回
29  }
```

运行程序，就可以看到在动态变化的标题栏图标。

26.3.2　实现图标动画

通过 DrawIcon()函数实现图标动画的原理及方式和标题栏动画图标的实现是一样的，其区别仅在于在此定时器处理函数中，会使用 DrawIcon()函数在指定位置显示指定索引处的图标。具体代码如下：

```
01  BOOL CMainFrame::DrawNextIcon()                        //绘制下一个图标
02  {
03      CDC* pDC = GetDC();                                //获取设备上下文
04      //定义图标数组
05      char* dwIcons[] = {
06          IDI_APPLICATION,IDI_ASTERISK,IDI_ERROR,
07          IDI_EXCLAMATION,IDI_HAND,IDI_INFORMATION,
```

```
08          IDI QUESTION,IDI WARNING,IDI WINLOGO};
09          m iIconCounter++;                               //图标计数变量
10          if(m iIconCounter >= 9)
11              m_iIconCounter =0;    //如果大于指定值，则归 0
12          //装载图标
13          HICON hIcon = LoadIcon(NULL,
14              MAKEINTRESOURCE(dwIcons[m iIconCounter]));
15          if (hIcon == NULL)
16              return false;            //判断装载结果
17          CRect rect;
18          GetClientRect(&rect);                           //获取工作区
19          //在指定区域绘制新图标
20          pDC->DrawIcon(rect.Width()/2, rect.Height()/2, hIcon);
21          return true;                                    //函数成功返回
22    }
```

上面代码会定时切换显示系统定义的图标，并通过 DrawIcon()函数在屏幕的中间位置显示图标。运行时，会看到屏幕中间的图标在不停地变换，运行效果如图 26-3 所示。

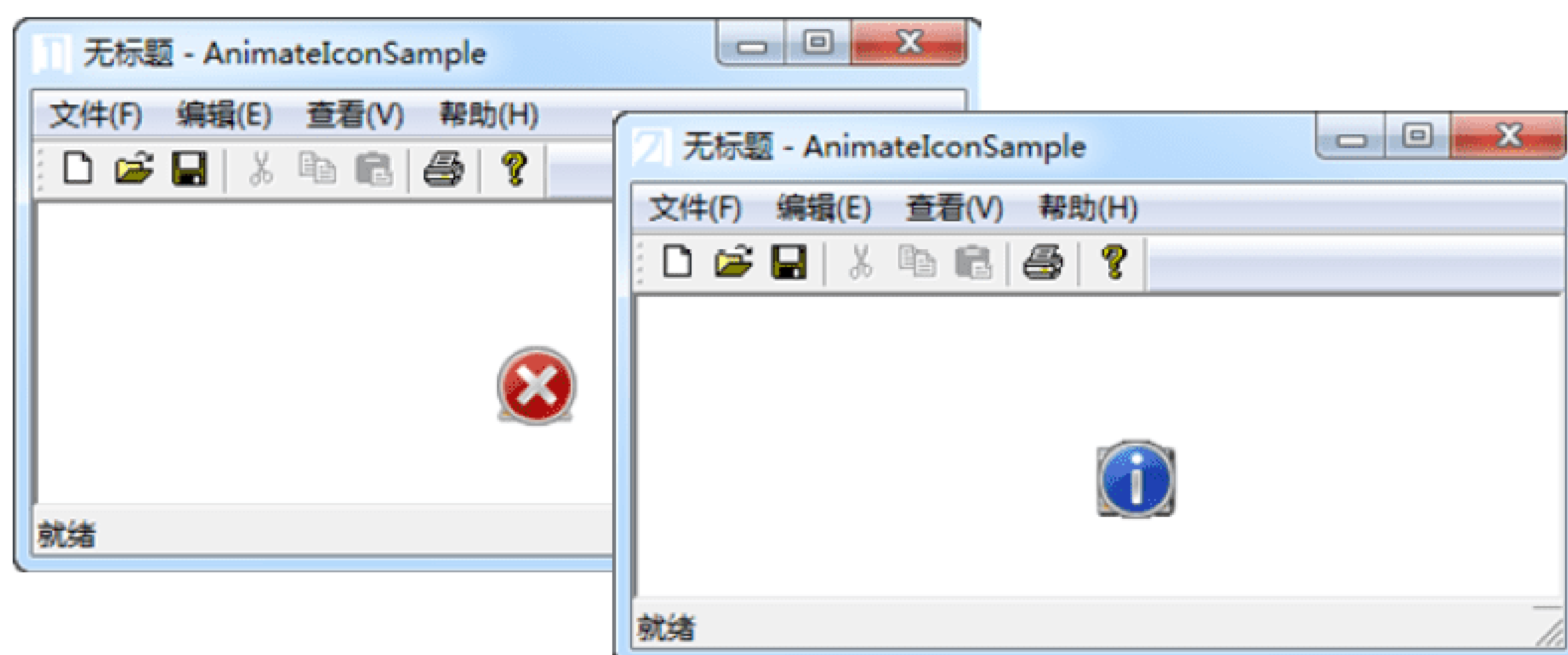


图 26-3　通过 DrawIcon 实现图标动画效果

26.3.3　系统托盘动态图标

通过定时器，可以实现系统托盘动态图标的效果。在程序启动时，首先设置图标列表，启动定时器，并设置托盘图标和托盘文字。定时器每次获取当前索引下的图标句柄，并调用 Shell_NotifyIcon()函数设置托盘上的图标为获取的新图标，这样，看上去系统托盘上显示的是动态图标。代码如下：

```
01  BOOL CMainFrame::SetSBImageList(int imageListID,
02      int iMaxIcons, COLORREF tc)
03  {                                                   //设置图标列表
04      if(iMaxIcons <= 0)
05          return false;            //如果最大个数小于或等于 0，则返回
06      m_iMaxSBIcon = iMaxIcons;                       //赋值保存图标个数
07      VERIFY(m SBImgList.Create(imageListID, 16, 1, tc));
08
09      hSBIcon = m SBImgList.ExtractIcon(0);           //获取图标
10      NOTIFYICONDATA nid;
11      ZeroMemory(&nid, sizeof(NOTIFYICONDATA));       //初始化内存
12      nid.cbSize = sizeof(NOTIFYICONDATA);            //赋值大小
13      nid.hWnd = m_hWnd;                              //赋值窗体句柄
```

```
14      nid.uID = IDI_ICON_APP;                          //赋值图标 ID
15      nid.uFlags = NIF_TIP | NIF_ICON | NIF_MESSAGE;   //赋值标记
16      nid.uCallbackMessage = NULL;                     //赋值回调消息
17      nid.hIcon = hSBIcon;                             //赋值图标句柄
18      strcpyn(nid.szTip, __TEXT("托盘图标测试!"), 64); //复制文本
19      Shell_NotifyIcon(NIM_ADD, &nid);                 //修改图标
20      return true;
21  }
```

上面的代码除了初始化图标列表外，调用 Shell_NotifyIcon()函数设置启动时托盘上的图标为图标列表中的第一个图标，并设置当鼠标滑过托盘图标时的提示文字为“托盘图标测试!”。下面的代码是定时器执行函数。

```
01  BOOL CMainFrame::ShowSBNextIcon()                   //显示下一个图标
02  {
03      //判断图标列表有效性
04      if(m_SBImgList.m_hImageList == NULL)
05          return false;
06      m_iSBIconCounter++;                             //索引计数增 1
07      if(m_iSBIconCounter >= m_iMaxSBIcon)
08          m_iSBIconCounter =0;
09      hSBIcon = m_SBImgList.ExtractIcon(m_iSBIconCounter);//获取图标
10      NOTIFYICONDATA nid;
11      ZeroMemory(&nid, sizeof(NOTIFYICONDATA));       //初始化内存
12      nid.cbSize = sizeof(NOTIFYICONDATA);            //设置结构大小
13      nid.hWnd = m_hWnd;                              //设置窗体句柄
14      nid.uID = IDI_ICON_APP;                         //设置图标
15      nid.uFlags = NIF_ICON;
16      nid.hIcon = hSBIcon;                            //设置图标句柄
17      Shell_NotifyIcon(NIM_MODIFY, &nid);             //设置图标
18      if (hSBPrevIcon)
19          DestroyIcon(hSBPrevIcon);                   //销毁句柄
20      hSBPrevIcon = hSBIcon;                          //保存图标句柄
21      return true;                                    //函数成功返回
22  }
```

上面代码会获取当前索引处的图标句柄，并通过 Shell_NotifyIcon()函数的 NIM_MODIFY 命令来修改托盘上的图标，同时修改当前图标索引。程序运行的效果如图 26-4 所示。

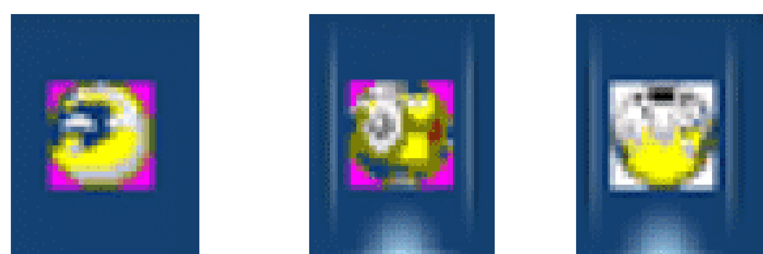


图 26-4　系统托盘动态图标运行效果

26.4　多媒体文件的播放

多媒体文件的格式有多种，本节介绍几种常见的多媒体文件的播放方法。主要包括如

何实现播放 GIF 动画、Flash 动画、VCD 文件和显示 JPEG 图像。读者通过本节的学习，应该掌握这些基本多媒体文件的播放方法。

注意：Visual C++ 6.0 提供和注册的一些 Active X 组件 Visual Studio 2010 并没有提供和注册，所以需要找到相关的组件文件并注册才可以在 Visual Studio 2010 中使用。本节的程序是在 Visual C++ 6.0 中演示的。示例同样可以在组件被注册了以后在 Visual Studio 2010 中使用。

26.4.1　播放 GIF 动画

GIF 文件是图形交换文件的格式，是由一组相隔指定间隔时间显示的图片组成。使用 GDI+可以播放 GIF 动画。为此，首先需要调用 GetFrameDimensionsCount()函数来获取 GIF 动画中具有的帧数数目，并通过 Image 对象的 GetPropertyItem()函数获取每帧图片之间的时间间隔。然后显示 GIF 文件，并设置当前有效的框架数据，根据获取的每帧的时间间隔，停顿一定的时间，继续显示。如此循环，就可以播放 GIF 动画了。具体代码如下：

```
01  void CPlayMultiMediaDlg::OnButtonPlaygif()          //播放 GIF 文件
02  {
03      Image image(L"C:\\byebye.gif");                 //定义 Image 对象
04      UINT uiCount = image.GetFrameDimensionsCount();//获取帧数
05      GUID *pDimensionIDs=(GUID*)new GUID[uiCount];
06      image.GetFrameDimensionsList(pDimensionIDs, uiCount);
07      UINT uiFrameCount=image.GetFrameCount(&pDimensionIDs[0]);
08      delete []pDimensionIDs;
09      UINT uiSize;
10      uiSize = image.GetPropertyItemSize(PropertyTagFrameDelay);
11      //获取帧延时长度
12      PropertyItem* pItem = (PropertyItem*)malloc(uiSize);
13      image.GetPropertyItem(PropertyTagFrameDelay, uiSize, pItem);
14      //获取属性项
15      GUID Guid = FrameDimensionTime;
16      CDC* pDC = GetDC();                              //获取设备上下文
17      while(true)                                      //依次处理每帧
18      {
19          Graphics gh(pDC->m_hDC);                 //hDC 是外部传入的画图 DC
20          gh.DrawImage(&image,0, 0, image.GetWidth(),
21              image.GetHeight());
22          //重新设置当前的活动数据帧
23          image.SelectActiveFrame(&Guid, uiCount++);
24          if(uiCount == uiFrameCount)
25              uiCount= 0;
26          //计算此帧要延迟的时间
27          long lPause = ((long*)(pItem->value))[uiCount];
28          Sleep(lPause);                           //停止指定长度的时间值
29      }
30  }
```

上面代码在画布的左上角播放 C:\byebye.gif GIF 动画。

26.4.2　播放 Flash 动画

虽然可以通过读取 Flash 文件格式的方式来播放 Flash 动画，但是 Windows 中提供了专门用于播放 Flash 动画的组件，可以通过使用此组件，简单地实现播放 Flash 动画的效果。具体方法如下。

（1）像插入其他 ActiveX 控件的方式一样，插入 Shockwave Flash Object 组件，它是一个 OCX 控件，名称为 Flash10b.ocx，如图 26-5 所示。

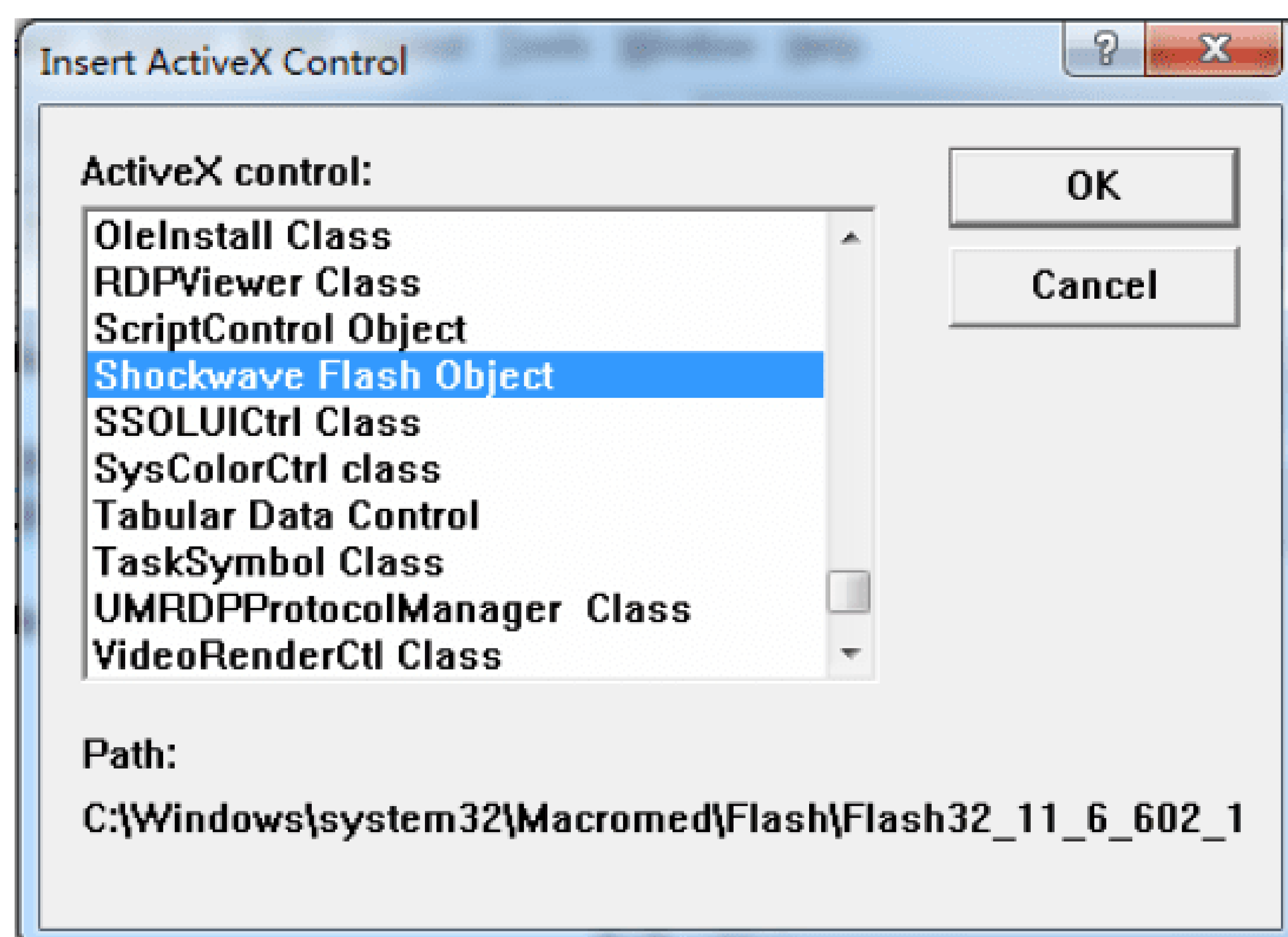


图 26-5　插入 Flash 组件

（2）在图 26-5 中，单击 OK 按钮。这样，对话框中就增加了 Flash 播放组件，调整到需要的大小和位置。使用前面章节介绍过的方法，添加与其对应的控件变量 m_shockWaveFlash。在添加时，如果工程中没有相应的类文件，会提示添加对应的类文件。

（3）在按钮的事件处理函数中，添加播放 Flash 的代码，代码如下：

```
01  void CPlayMultiMediaDlg::OnButtonPlayflash()        //播放 Flash
02  {
03      m_shockWaveFlash.SetMovie("C:\\reading1.swf"); //设置 Flash 文件名
04      m_shockWaveFlash.Play();                       //播放 Flash
05  }
```

上面代码会装载完整文件名为 C:\ reading1.swf 的 Flash 文件，并调用 Flash 播放组件的 Play()函数，播放 Flash 动画。

（4）编译、链接并生成运行程序，单击“播放 Flash 动画”按钮，即会在对话框中播放 Flash 动画。

26.4.3　播放 VCD

VCD（Video Compact Disc）即视频压缩盘片，是较早的一种多媒体格式标准，是由索尼、飞利浦、JVC、松下等厂商联合于 1993 年提出的。VCD 格式分为两种 MPG 格式或 DAT 格式，MPG 格式用于保存电脑编辑的 VCD，DAT 格式用于刻录成光盘后的格式。使用 Windows Media Player 组件可以容易地实现播放 VCD。具体方法如下（在 Visual C++ 6.0

中）。

（1）像插入其他 ActiveX 控件的方式一样，插入 Windows Media Player 组件，如图 26-7 所示。

（2）在图 26-6 中，单击 OK 按钮。这样，对话框中就增加了 Windows Media Player 播放组件，调整到需要的大小和位置。使用前面章节介绍过的方法，添加与其对应的控件变量 m_wmPlayer。在添加时，如果工程中没有相应的类文件，会提示添加对应的类文件，如图 26-7 所示，单击 OK 按钮。

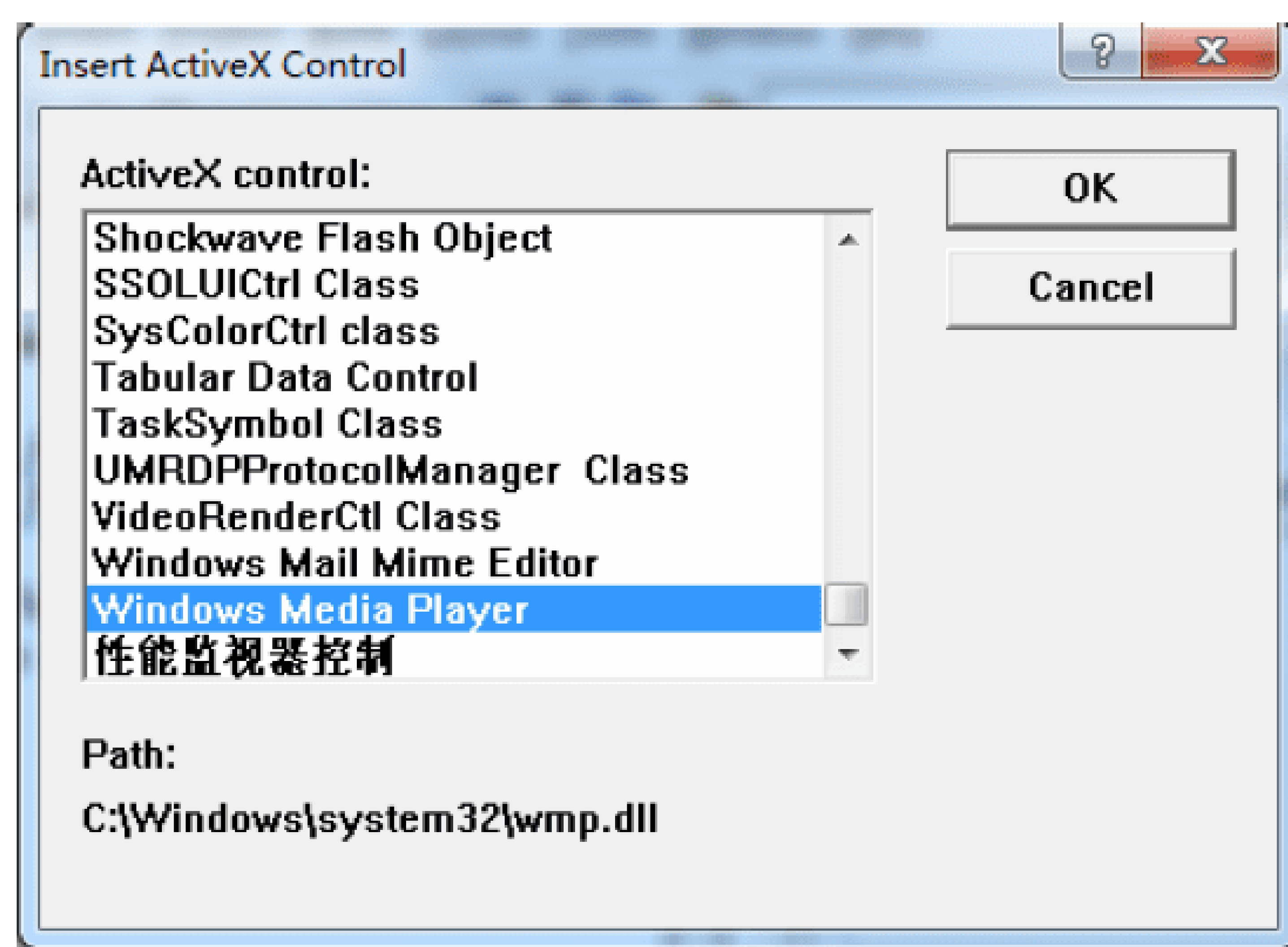


图 26-6　插入 Windows Media Player 组件

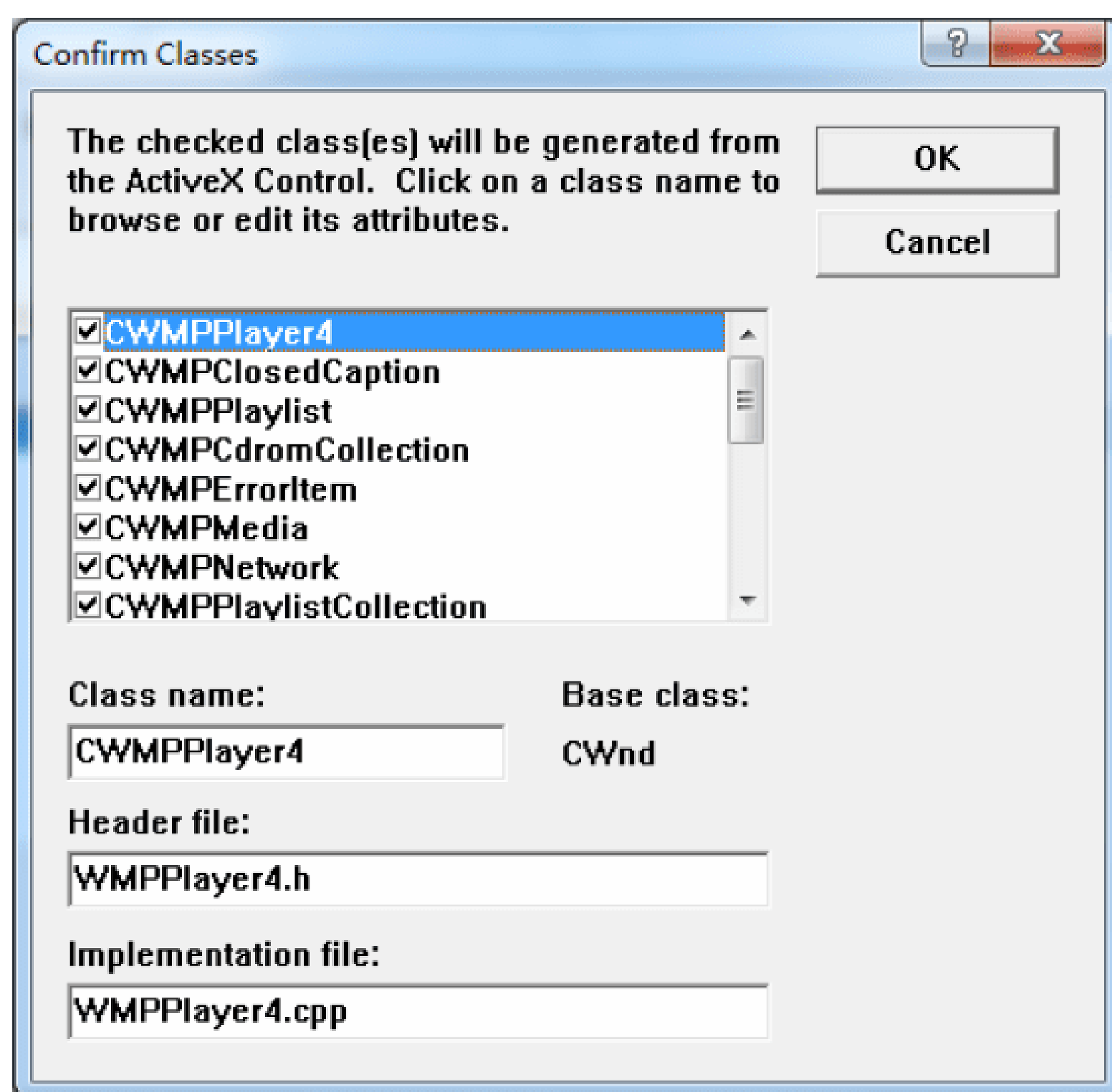


图 26-7　确认添加类对话框

（3）在按钮的事件处理函数中，添加播放 VCD 的代码，代码如下：

```
01  void CPlayMultiMediaDlg::OnButtonPlaycd()       //播放 VCD 文件
02  {
03      //设置 VCD 文件名
04      m_wmPlayer.SetUrl("E:\\LLN\\MPEGAV\\AVSEQ01.DAT");
05  }
```

上面代码调用 m_wmPlayer 组件的 SetUrl()函数播放 E:\LLN\MPEGAV\AVSEQ 01.DAT 文件。

（4）编译、链接并生成运行程序，单击“播放 VCD”按钮，即会在组件对话框中播放 VCD。

26.4.4　显示 JPEG 图像

JPEG（Joint Photograhic Experts Group）是一种高压缩比的图像格式，是目前最常用的图像格式之一。本小节介绍如何在对话框中显示 JPEG 图像。分为以下几个步骤。

（1）调用 CreateFile()函数打开 JPG 文件，获取文件的大小，调用 GlobalAlloc()函数分配图像占用的内存空间。

（2）调用 ReadFile()函数读取文件内容放置到内存中，并调用 CreateStreamOnHGlobal()

函数根据文件内容创建数据流。

（3）调用 OleLoadPicture()函数将数据流载入 PICTURE 对象中，根据 JPEG 图像的高和宽的比例以及画布的高和宽，计算可以显示的图像范围。调用 PICTURE 对象的 Render()函数在画布上渲染图像。具体代码如下：

```
01  void CPlayMultiMediaDlg::OnButtonShowjpeg()        //显示 JPEG 图像
02  {
03      CDC*  pDC=GetDC();                              //获取设备上下文
04      LPPICTURE  gpPicture = NULL;
05      CString fileName = "C:\\BeautyGirl.JPG";        //定义文件路径变量
06      HANDLE hFile = CreateFile(fileName, GENERIC_READ,
07          0,  NULL,OPEN_EXISTING, 0, NULL);           //创建文件句柄
08      f  (hFile==INVALID_HANDLE_VALUE)
09          return;
10      DWORD  dwFileSize = GetFileSize(hFile, NULL);  //取得文件大小
11      if (dwFileSize == -1)
12          return;
13      LPVOID pvData=NULL;
14      HGLOBAL hGlobal= GlobalAlloc(GMEM_MOVEABLE, dwFileSize);
15      //根据文件大小分配内存
16      if (hGlobal != NULL)
17          pvData=GlobalLock(hGlobal); //锁定存储区
18      if (pvData  == NULL)
19          return;
20      DWORD dwBytesRead = 0;
21      BOOL bRead= ReadFile(hFile, pvData, dwFileSize,
22          &dwBytesRead, NULL);
23      //读取文件
24      GlobalUnlock(hGlobal);                          //释放存储区
25      CloseHandle(hFile);                             //关闭文件句柄
26      if (!bRead)
27          return;
28      LPSTREAM pstm=NULL;
29      HRESULT hr=CreateStreamOnHGlobal(hGlobal,
30          true, &pstm); //创建数据流
31      if (!SUCCEEDED(hr))
32          return;
33      if (gpPicture)
34          gpPicture->Release();
35      hr = OleLoadPicture(pstm, dwFileSize, false,
36          IID_IPicture, (LPVOID*)&gpPicture);         //装载图片
37      if (!SUCCEEDED(hr))
38          return;
39      pstm->Release();
40      HDC hdc=pDC->GetSafeHdc();                      //获取上下文
41      if (gpPicture)                                  //如果图片有效
42      {
43          long  hmWidth, hmHeight;                    //取得图片的宽和高
44          gpPicture->get_Width(&hmWidth);
45          gpPicture->get_Height(&hmHeight);
46          int inch = 2540;                            //宽、高转换为像素
47          int nWidth = MulDiv(hmWidth,
48              GetDeviceCaps(hdc, LOGPIXELSX), inch);
49          int  nHeight = MulDiv(hmHeight,
50              GetDeviceCaps(hdc, LOGPIXELSY),inch);
51          RECT rc;
52          GetClientRect(&rc);                         //取得客户区
```

```
53          int width = rc.right - rc.left;
54          int height = rc.bottom - rc.top;
55          gpPicture->Render(hdc, 0, 0,
56              (int)height*hmWidth/hmHeight, height,
57              0, hmHeight, hmWidth, -hmHeight, &rc);   //渲染图片
58      }
59  }
```

上面的代码会在对话框中尽可能大地显示文件名为 C:\BeautyGirl.JPG 的 JPEG 图像。

26.5　本 章 小 结

本章主要讲解了有关声音和动画的操作，重点介绍了音频操作、多媒体播放和应用以及动画效果的实现。本章的难点是多媒体文件的播放。第 27 章将具体介绍 DirectX 中的图形开发。

26.6　习　　题

1．扩展 26.2.3 小节讲解的示例程序，使得程序可以绘制圆角矩形和椭圆形。

【思路】绘制椭圆形的函数是 Ellipse()，绘制圆角矩形的函数是 RoundRect()，具体的使用方法可以在 MSDN 中查找到。

2．对于 26.3 节所讲的示例做这样的修改：插入一个自己的图标，让此图标和其他图标一同在系统托盘上循环显示。

【思路】理解 26.3 节所讲解示例的运行机理，恰当地修改部分代码。

第 27 章　DirectX 图形开发

DirectX 是微软为了提高系统性能而提供的一组开发组件，可以高效地完成对音频、视频、输入设备和网络等各方面的操作。DirectX 在各个方面的处理都非常优秀，只是其在图形方面尤为突出，所以，在其他几个方面的应用中往往被人们忽略。因此，建议读者根据需要可以研究学习 DirectX 的其他方面的开发组件。本章仅讲述 DirectX 图形方面的开发。

27.1　DirectX SDK

要使用 DirectX 组件进行程序开发，首先需要安装 DirectX SDK，然后需要在 Visual Studio 2010 环境中配置 DirectX SDK 的工作参数，使 Visual Studio 2010 和 DirectX 组件可以集成在一起。这样，在 Visual Studio 2010 中才可以自如地使用 DirectX 组件。本节将介绍如何安装和配置 DirectX SDK 组件。

27.1.1　DirectX SDK 的安装

DirectX SDK 的安装步骤如下。

（1）获取 DirectX SDK 的安装包。可以通过微软的官方网站下载 DirectX SDK 的安装程序。本书中使用的是 DXSDK_Jun10.exe 版本，此版本的安装包大约是 571MB。

（2）双击安装程序，打开 DirectX SDK 的安装欢迎对话框，如图 27-1 所示。

（3）单击“下一步”按钮，弹出许可对话框，如图 27-2 所示。

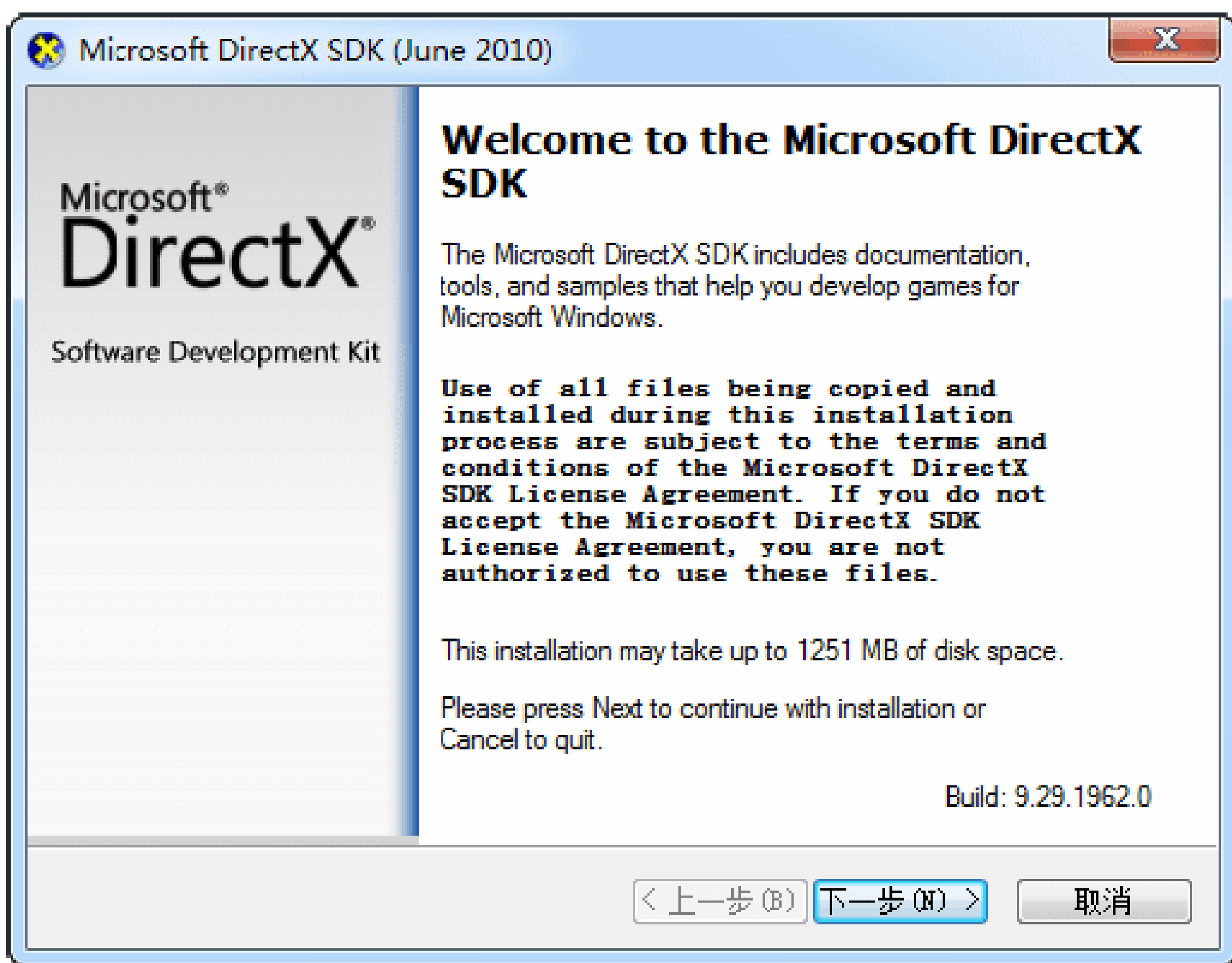


图 27-1　DirectX SDK 安装步骤之欢迎界面



图 27-2　DirectX SDK 安装步骤之许可对话框

（4）选择 I accept the terms in the license agreement 单选按钮，单击“下一步”按钮，

弹出安装选项对话框，如图 27-3 所示。

（5）选择要安装的组件，单击“下一步”按钮，开始安装程序，如图 27-4 所示。

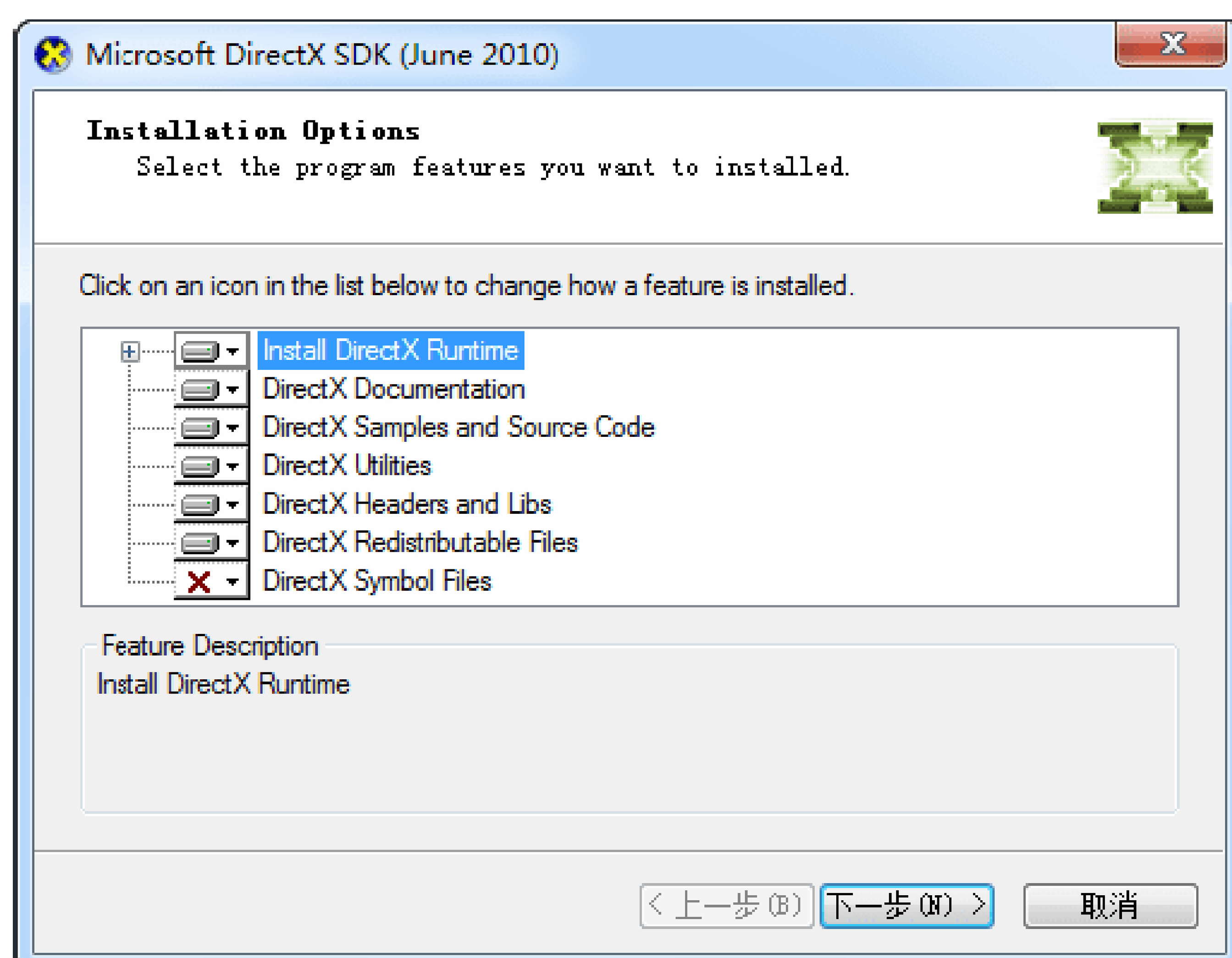


图 27-3　DirectX SDK 安装步骤之安装选项对话框

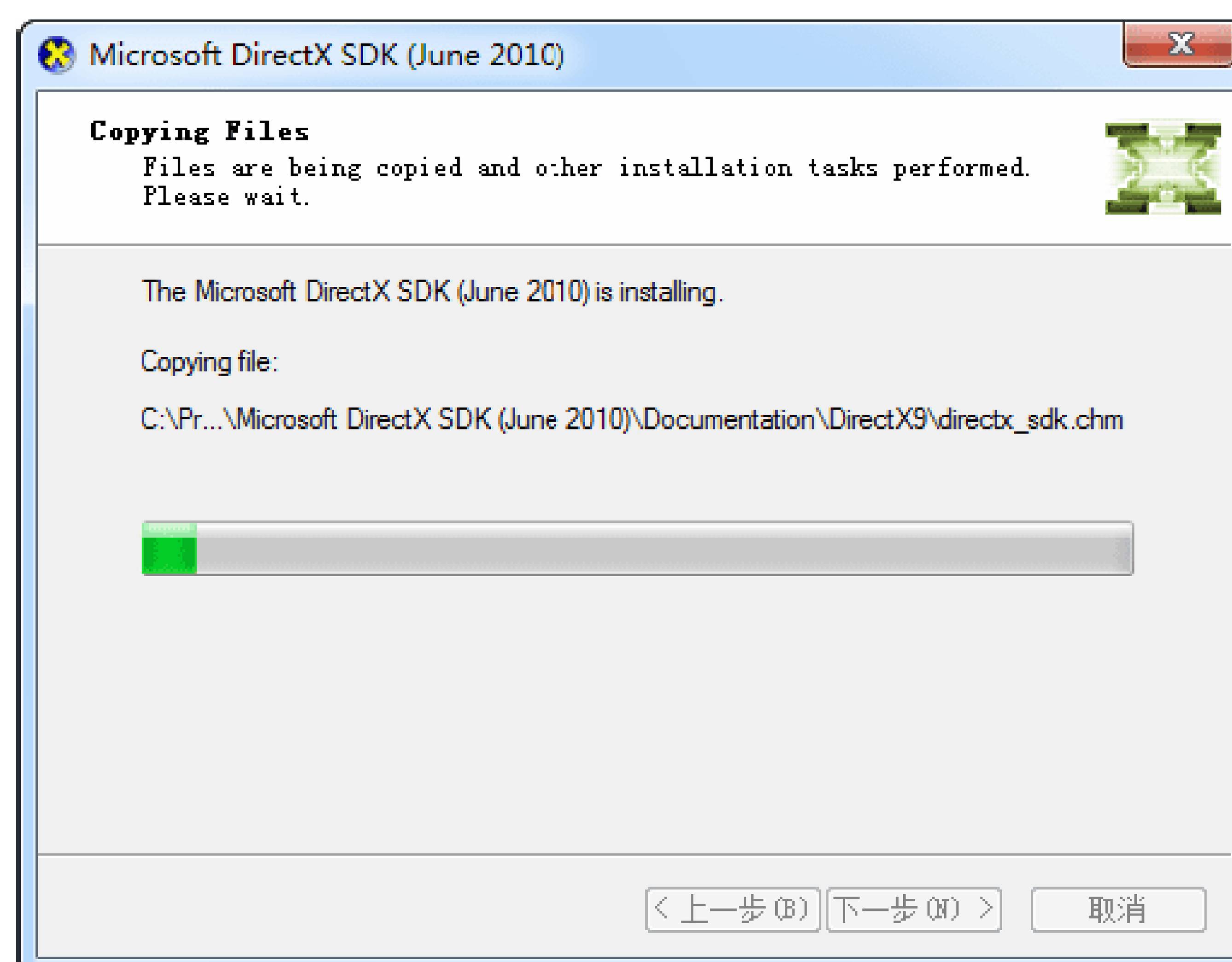


图 27-4　DirectX SDK 安装步骤之安装进度对话框

（6）安装完成后，会弹出提示对话框，如图 27-5 所示。单击“完成”按钮，DirectX SDK 就安装完成了。

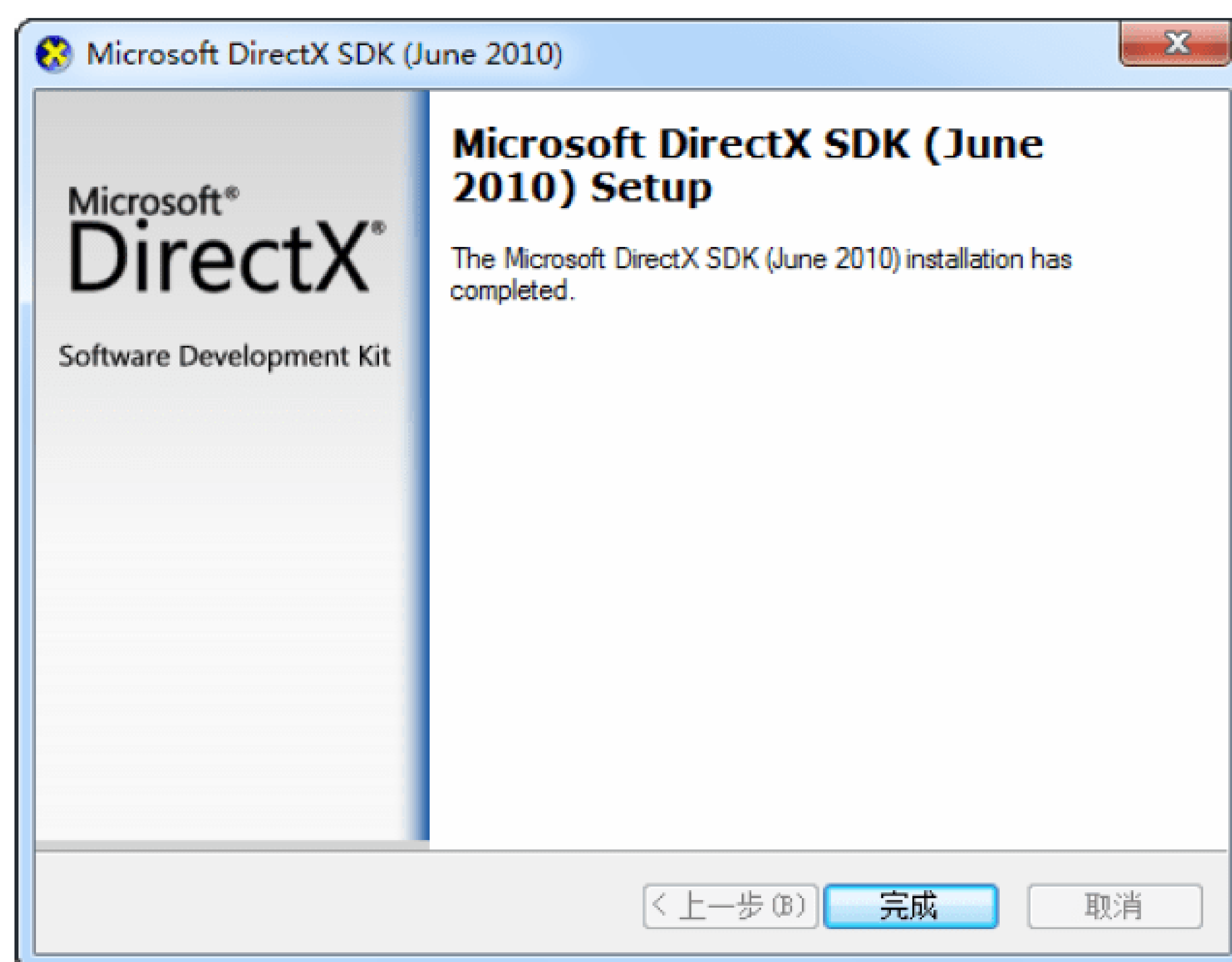


图 27-5　DirectX SDK 安装步骤之安装完成对话框

27.1.2　Visual Studio 2010 中的相应设置

安装完成后，要在 Visual Studio 2010 中使用 DirectX 组件，还需要在 Visual Studio 2010 中对 DirectX SDK 进行配置。配置的过程如下。

（1）启动 Visual Studio 2010，按照前面章节介绍的方法创建工程，本章创建 SDI 工程 DXTest。

（2）鼠标右击“解决方案资源管理器”中的项目 DXTest，在弹出的快捷菜单中选择“属性”命令，弹出如图 27-6 所示的 “DXTest 属性页” 对话框。

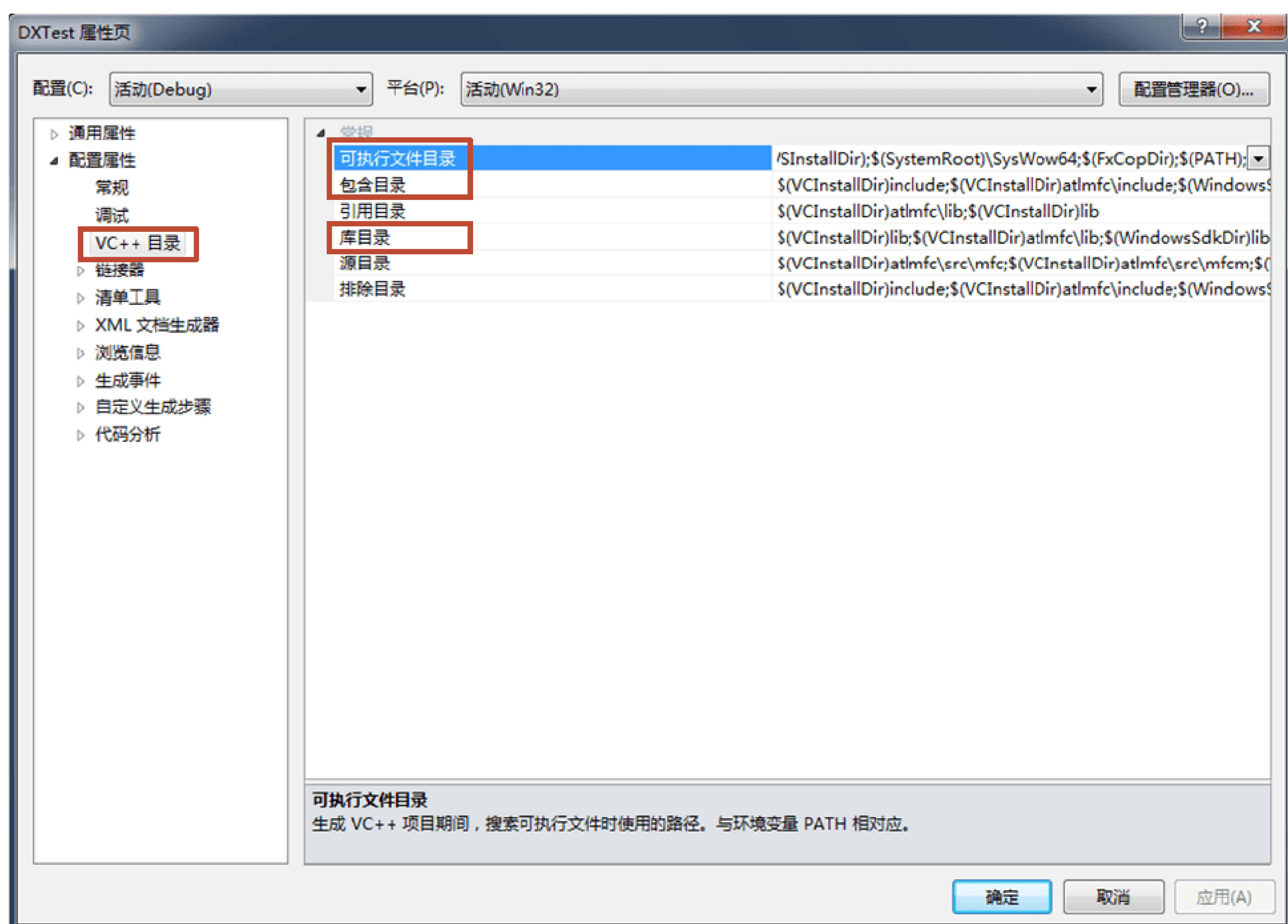


图 27-6　“DXTest 属性页”对话框

（3）选择“VC++目录”。修改“可执行文件目录”、“包含目录”和“库目录”，即分别添加新的路径：$(DXSDK_DIR)Utilities/Bin/x86、$(DXSDK_DIR)Include 和 $(DXSDK_DIR)Lib/x86，如图 27-7 所示。

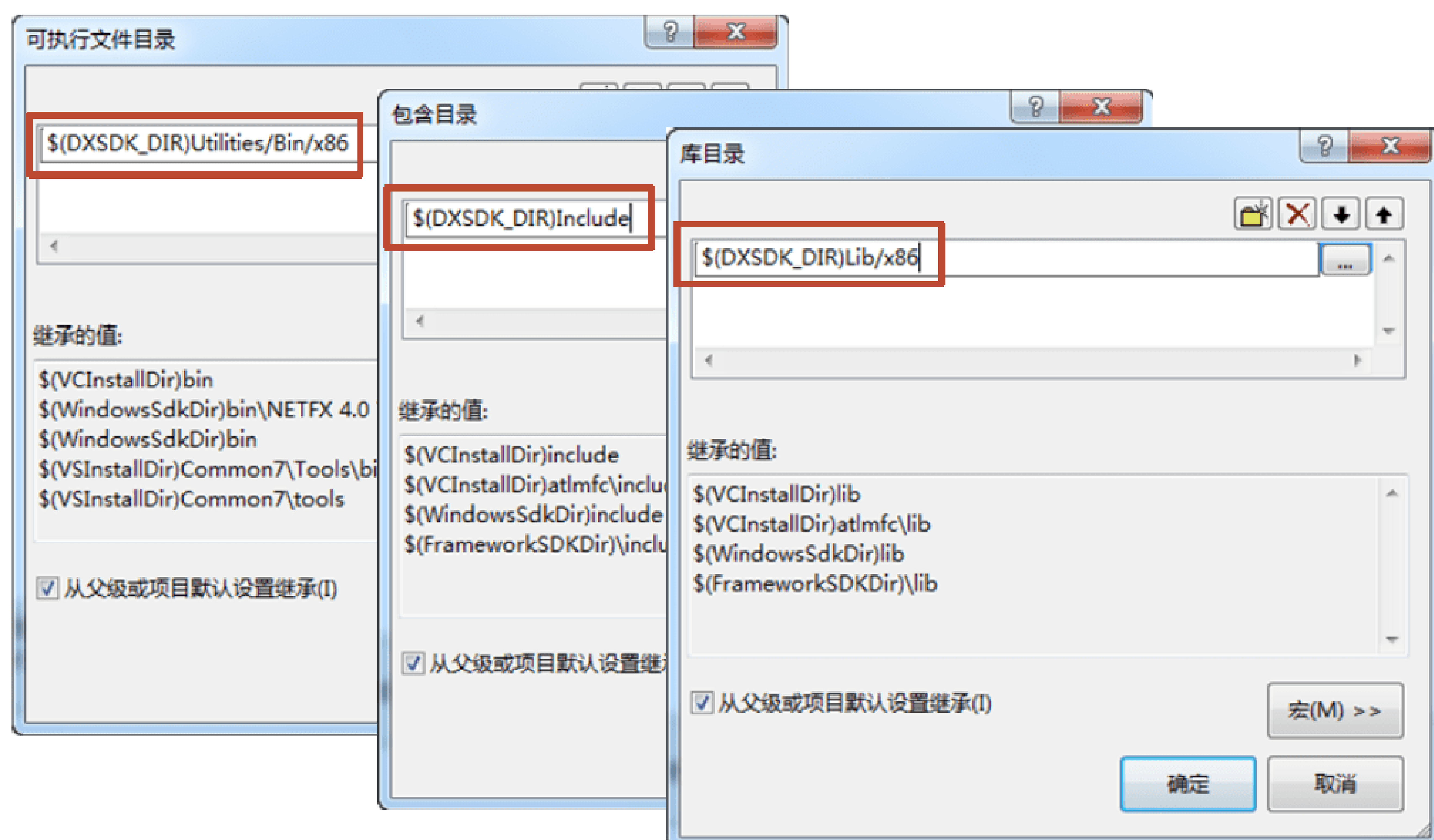


图 27-7　添加新的路径

（4）选择“链接器”|“输入”命令，如图 27-8 所示，修改“附加依赖项”，即添加 d3dx9d.lib、d3dx10d.lib、d3d9.lib 和 winmm.lib。如图 27-9 所示。

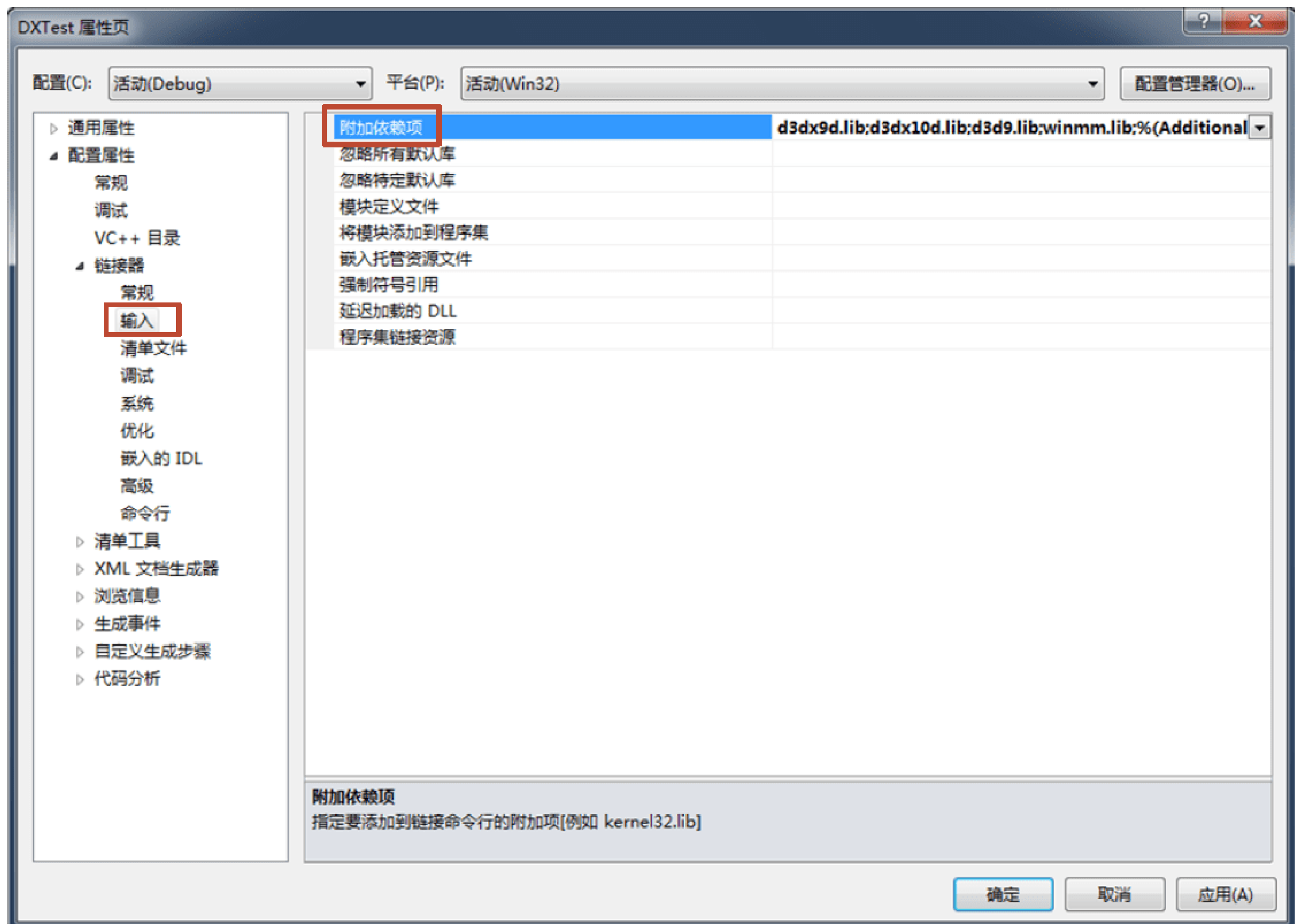


图 27-8　修改“附加依赖项”

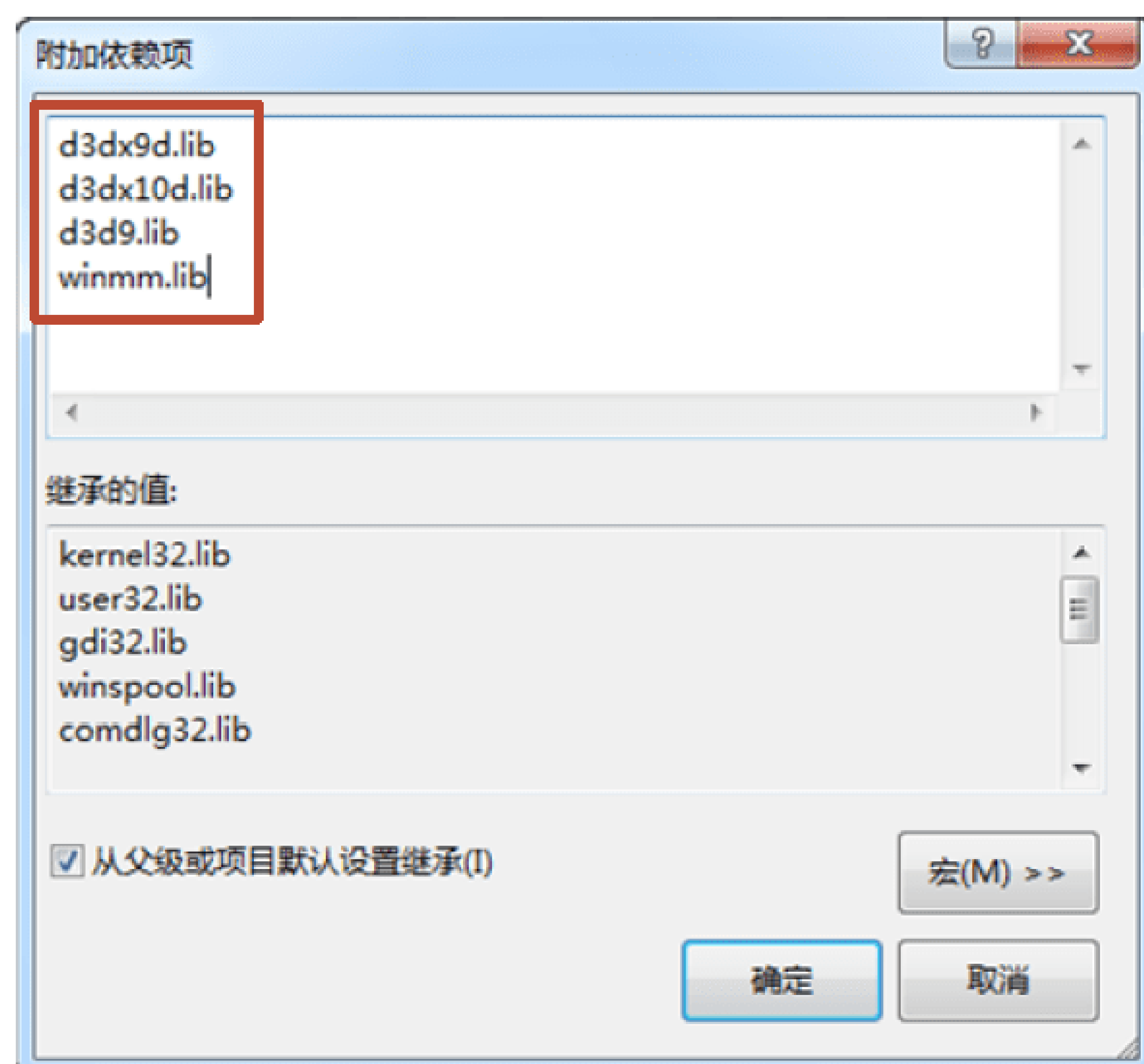


图 27-9　添加静态库

（5）在代码中使用#include 预定义语句引入用到的 DirectX 头文件，并加入使用 DirectX 实现的功能的代码，编译、链接、调试和运行即可。这样就完成了 Visual Studio 2010 中 Direct SDK 的配置。

27.2　DirectX 9.0 介绍

DirectX 是微软推出的一套在 Windows 平台上进行多媒体应用程序开发的 API 接口。DirectX 9.0 是在 Direct X 的早期版本上发展而来的，其具有多种新特性。但是其使用方法与早期版本类似。本节将简单地介绍 DirectX 的组件，并以使用 DirectX 为例讲解如何在 Visual Studio 2010 中使用 COM 组件。

27.2.1　DirectX 组件介绍

DirectX 是一个辅助的图形应用程序接口软件，不仅可以提高系统性能，还可以实现与硬件设备无关的编程。主要包括两部分，一部分是运行库，使用 DirectX 的应用程序必须有运行库才可以正常执行；另一部分是开发库，即 DirectX SDK，27.1 节讲过其安装和配置，必须安装 DirectX SDK 才可以编译生成 DirectX 程序。

DirectX 包含两层含义，Direct 表示直接的，X 表示多个方面。二者综合在一起就是直接集成多种操作的组件，即为 Windows 下多媒体编程的性能提高提供了一组组件，涉及到音频、视频、系统、网络和输入等多个方面的内容。主要包括以下内容。

- ❑ DirectX Graphics 组件：集成了 DirectDraw 和 Direct3D 技术。其中 DirectDraw 主要通过对显卡内存和系统内存的直接操作来实现二维图形图像的加速。Direct3D 主要提供用于绘制三维图像的硬件接口，是开发三维游戏的基础。
- ❑ Direct Show 组件：主要为多种格式的多媒体文件的回放、音视频采集等高性能要求的多媒体应用提供接口。DirectX 中还包含 Direct Media Objects 组件，提供了 Direct Show Filter 的简化模型和更简单的流数据处理。
- ❑ Direct Music 组件：主要提供 MIDI 音乐合成和播放方面的开发接口，与 Direct Show 组件的侧重点是不同的，主要提供有关音乐方面的编程。
- ❑ Direct Sound 组件：主要提供音频捕捉、回放、处理、硬件加速和访问声卡等有关音频的开发接口。
- ❑ Direct Play 组件：主要提供网络游戏的通信和组织功能。
- ❑ Direct Input 组件：主要提供对输入输出设备的支持，如通过此组件可以操作鼠标、键盘和游戏杆等各种设备。
- ❑ Direct Setup 组件：提供自动安装 DirectX 组件的开发接口。

从上述可以看出，DirectX 中主要设计音频、视频、图形图像、输入设备和网络通信等方面的支持，这些都是开发多媒体应用程序需要的功能，使用 DirectX 系列的组件开发程序，可以大大简化开发人员的工作量。

27.2.2　使用 COM

DirectX 采用 COM 标准，它是一组 COM 组件。因此，要使用 DirectX 组件，必须要掌握 COM 组件的使用方法。COM 本身是一组规范，而不是具体实现，在 VC 中，可以将

COM 组件看作 C++类，而 COM 组件中的接口就是 C++的纯虚类。在使用 DirectX 时，会用到两方面的 COM 技术，一种是作为 COM 组件的客户程序，此种情况，只需要掌握 COM 组件的使用知识即可；另一种是开发 COM 组件，需要掌握如何实现 COM 组件。在使用 COM 组件前，首先了解下节介绍的几个概念。

任何组件或接口必须从 IUnknown 接口继承，并且具有唯一的 GUID 标识符。组件通过类工厂来实现，其实现了 IClassFactory 接口，并在 CreateInstance()函数中使用 new 操作符创建 COM 组件类的对象实例。下面是创建 Direct3D 设备的代码。

```
IDirect3D9*  m_d3d = Direct3DCreate9(D3D_SDK_VERSION);
```

从上面可以看出，使用 COM 组件，可以像调用其他接口函数一样来调用组件的接口函数。

27.3　DirectX 图形开发基本概念

DirectX 图形开发组件中集成了用于二维图像开发的 DirectDraw 组件和用于三维图像开发的 Direct3D 组件。在使用 Direct3D 组件进行三维图像开发时，需要了解从三维现实世界投影到二维屏幕的一些基本概念，本节将介绍 DirectX 图形开发的基本概念。

27.3.1　世界坐标系

世界坐标系是三维物体在现实空间中的坐标系统。世界坐标系是由 3 条互相垂直并相交的直线组成，相交点称为坐标原点，3 条直线是 3 条坐标轴。在屏幕上，x 轴是水平向右的，y 轴是垂直向上的，z 轴是指向用户的，也可以将其看作左手笛卡尔坐标系统。在绘图过程中，世界坐标系的原点和坐标轴是不变的。每个现实世界中的三维物体都具有在世界坐标系中的坐标，即一组三元坐标（x，y，z），分别表示物体距离世界坐标系原点的水平偏移、垂直偏移和深度偏移。在三维空间中三维物体可以发生运动和变形，如旋转、平移等，此过程称为世界变换。

27.3.2　摄影坐标系

现实世界中物体的坐标是三维的，要将其显示在二维的屏幕上，就需要将三维的世界坐标系转换成二维的摄影坐标系。3D 绘图的过程类似于生活中的摄影，摄影是将现实世界中的物体的轮廓、样式和颜色拍摄下来，显示在二维的照片上。而在屏幕上绘制 3D 图形，是将现实世界中的 3D 场景中的坐标转换成平面的二维坐标，并结合灯光、材质、纹理贴图和视口变换等处理，计算出在二维屏幕上的显示坐标值和对应的颜色值，并在二维屏幕上绘制出 3D 图形。

所谓摄影坐标系，就是在进行从三维世界坐标系转换到屏幕二维坐标系过程中，首先需要摆放虚拟摄影机的位置，在当前摄影机的位置，拍摄三维场景，会生成与当前角度相符的图像，此时使用的坐标系为摄影坐标系。这也是进行转换的第一步。摄影坐标系是以虚拟摄影机的位置为坐标原点，虚拟摄影机的观察方向作为坐标轴，现实物体的坐标称为摄影坐标，从世界坐标到摄影坐标的转换过程称为取景变换。

27.3.3　剪裁和透视投影

在架设好虚拟摄影机，对三维场景进行取景后，因为摄影机进行拍摄时是有角度的，会以圆锥体的方式在一定的范围内取景，所以，会照射三维物体的一部分，此过程称为剪裁。剪裁过程会将摄影机覆盖不到的物体的部分剪裁掉。

经过剪裁，拍摄完三维场景后，物体的坐标就从世界坐标转换为摄影坐标，下一步就是将三维物体的取景投影到二维表面上，即将当前三维的摄影坐标投影到二维坐标，此过程类似于拍照过程中胶片的曝光过程，称为透视投影变换。此时以胶片中心为参考原点，称为投影坐标系，物体在投影坐标系中的坐标称为投影坐标。

27.3.4　视口变换和像素的光栅显示

物体在投影坐标中的坐标为浮点坐标，通过屏幕显示区域，将投影坐标系中的浮点坐标转换为屏幕上的像素坐标的过程称为视口变换。经过此过程转换后的坐标称为屏幕坐标，单位是像素。屏幕坐标除了与物体的投影坐标相关外，还与使用的屏幕显示区域的范围相关。如果定义视口的大小为 800×600，而投影坐标为（1.0f，0.5f），则经过视口变换得到的屏幕坐标为（800，300）；如果定义视口的大小为 1024×768，而投影坐标为（1.0f，0.5f），则经过视口变换得到的屏幕坐标为（1024，384）。此处视口可以理解为要显示三维场景的屏幕范围。

现实物体经过这一系列变换，从世界坐标转换成屏幕坐标后就可以进行光栅显示了。其变换过程如图 27-10 所示。

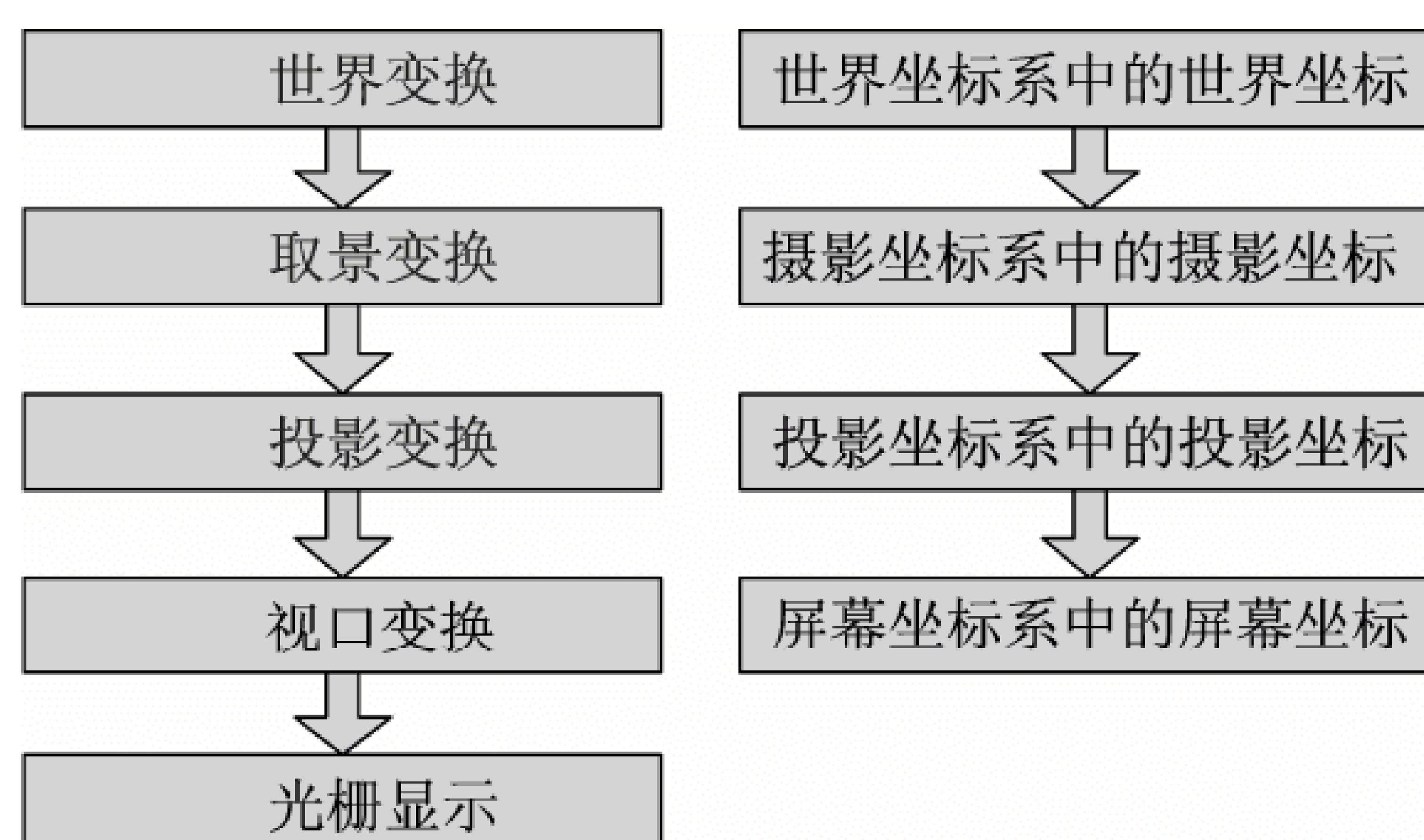


图 27-10　3D 坐标变换过程

虽然物体从现实的世界坐标转换为二维的屏幕坐标的变换比较复杂，但是，使用 DirectX Graphics 渲染 3D 图像时，只需要设置好顶点变换矩阵和视口信息，就可以进行图像顶点的坐标变换了。Direct3D 封装了其他操作，封装了复杂的变换过程，简化了开发工作量。

在 Direct3D 中使用 Direct3D 设备对象的 SetTransform()函数来设置顶点变换矩阵，其函数原型为：

```
HRESULT SetTransform(
  D3DTRANSFORMSTATETYPE State,              //指定要设置的变换矩阵类型
  CONST D3DMATRIX * pMatrix);               //要设置的变换矩阵
```

如果函数操作成功，返回 D3D_OK；如果参数无效，则返回 D3DERR_INVALIDCALL。D3DTRANSFORMSTATETYPE 枚举类型定义了变换矩阵的类型，其有效取值如下。

- ❑ D3DTS_VIEW：表示变换矩阵为视口变换矩阵。
- ❑ D3DTS_PROJECTION：表示变换矩阵为投影变换矩阵。
- ❑ D3DTS_TEXTURE0～D3DTS_TEXTURE7：表示变换矩阵为纹理变换矩阵。
- ❑ D3DTS_FORCE_DWORD：表示强制将变换矩阵编译为 32 位。

256～511 的变换类型值为世界变换预留，当使用 D3DTS_WORLDMATRIX 和 D3DTS_WORLD 宏进行变换时，可以使用其中的变换类型来指定。

在 Direct3D 中使用 Direct3D 设备对象的 SetViewport()函数来设置视口信息，其函数原型为：

```
HRESULT SetViewport(CONST D3DVIEWPORT9 * pViewport);
```

其中，pViewport 参数是指向 D3DVIEWPORT9 结构的指针，用于指定要设置的视口参数。如果函数操作成功，返回 D3D_OK；如果参数无效，如 pViewport 无效或其中定义的区域在渲染目标上不存在，则返回 D3DERR_INVALIDCALL。

三维场景中的物体的屏幕坐标获取并设置好视口后，就可以对其进行光栅显示。所谓光栅，是指纵横相交的线做成的小格，小格小到一定范围，就可以将其看作点。计算机中的屏幕显示就是对屏幕上的栅格点的显示，每个点称为一个像素，每个像素都有其对应的颜色。为每个栅格点渲染颜色，也称为像素的光栅显示。因此，对三维场景中的物体的显示，就是对其坐标点对应的像素点的颜色进行渲染。当物体的所有像素点的光栅显示都完成后，整个物体就在屏幕上显示出来了。在后面几节中会介绍如何完成在屏幕上对像素进行光栅显示。

27.3.5　显示卡的 3D 渲染管道线

Direct3D 组件通过提供显示卡的 3D 渲染管道线对上面的各个变换提供了封装，读者使用 3D 渲染管道线可以实现现实世界中的三维物体到二维屏幕的显示。具体流程如图 27-11 所示。

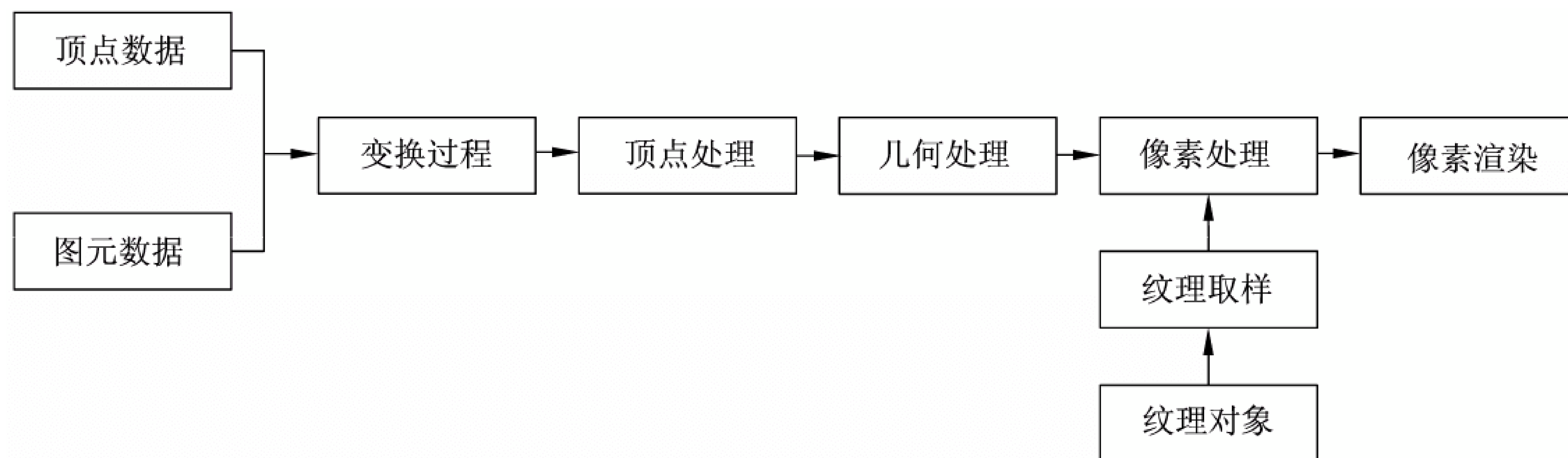


图 27-11　显示卡的 3D 渲染管道线

- ❑ 顶点数据：是存储在顶点内存中的对照模型的顶点数据，使用 IDirect3Dvertex-Buffer9 接口来表示。
- ❑ 图元数据：表示几何图元，包括点、线、三角形和多边形等形状，这些形状使用顶点索引缓冲区来引用顶点数据缓冲区中的顶点数据，使用 IDirect3Dindex-Buffer9 接口来表示。

- ❑ 变换过程：转换图元、位图和网格补丁为顶点位置，并在顶点缓冲区中存储。
- ❑ 顶点处理：对顶点缓冲区中的顶点数据进行 Direct3D 变换。
- ❑ 几何处理：对变换后的顶点进行剪裁、背面剔除、属性计算和光栅计算。
- ❑ 纹理对象：使用 IDirect3DTexture9 接口对 Direct3D 表面进行纹理化贴图。
- ❑ 纹理过滤：对纹理值进行纹理过滤。
- ❑ 像素处理：使用几何数据对输入的顶点和纹理数据进行像素着色，输出像素颜色值。
- ❑ 像素渲染：使用透明度、深度、模板测试、Alpha 混合或雾化等对像素进行渲染，最后生成要在屏幕上显示的像素的颜色值。

通过上面讲述的 3D 渲染管道线的流程，即可以完成三维图像到二维显示的数据变换。从 27.4 节开始，将以具体的实例讲解如何使用 DirectX 进行图形开发。

27.4　基本三角形面的绘制

DirectX Graphics 提供 Direct3D 组件和 D3DX 扩展组件，可以统一处理二维和三维的图形渲染，而不再需要使用早期版本中的 DirectDraw 组件进行二维图形的渲染。本节就以基本三角形面的绘制为例，讲解如何使用 DirectX Graphics 组件进行二维图形的处理。

27.4.1　DirectX Graphics 基本应用架构

要使用 DirectX Graphics 组件渲染图形，首先需要创建 IDirect3D9 接口对象，然后创建 Direct3D 设备，利用此 Direct3D 设备的接口方法控制管道流水线进行图形的渲染。使用 DirectX Graphics 组件的应用程序与使用 GDI 或 GDI+的应用程序在结构上略有不同，但是它们使用的底层接口都是设备驱动接口，如图 27-12 所示。

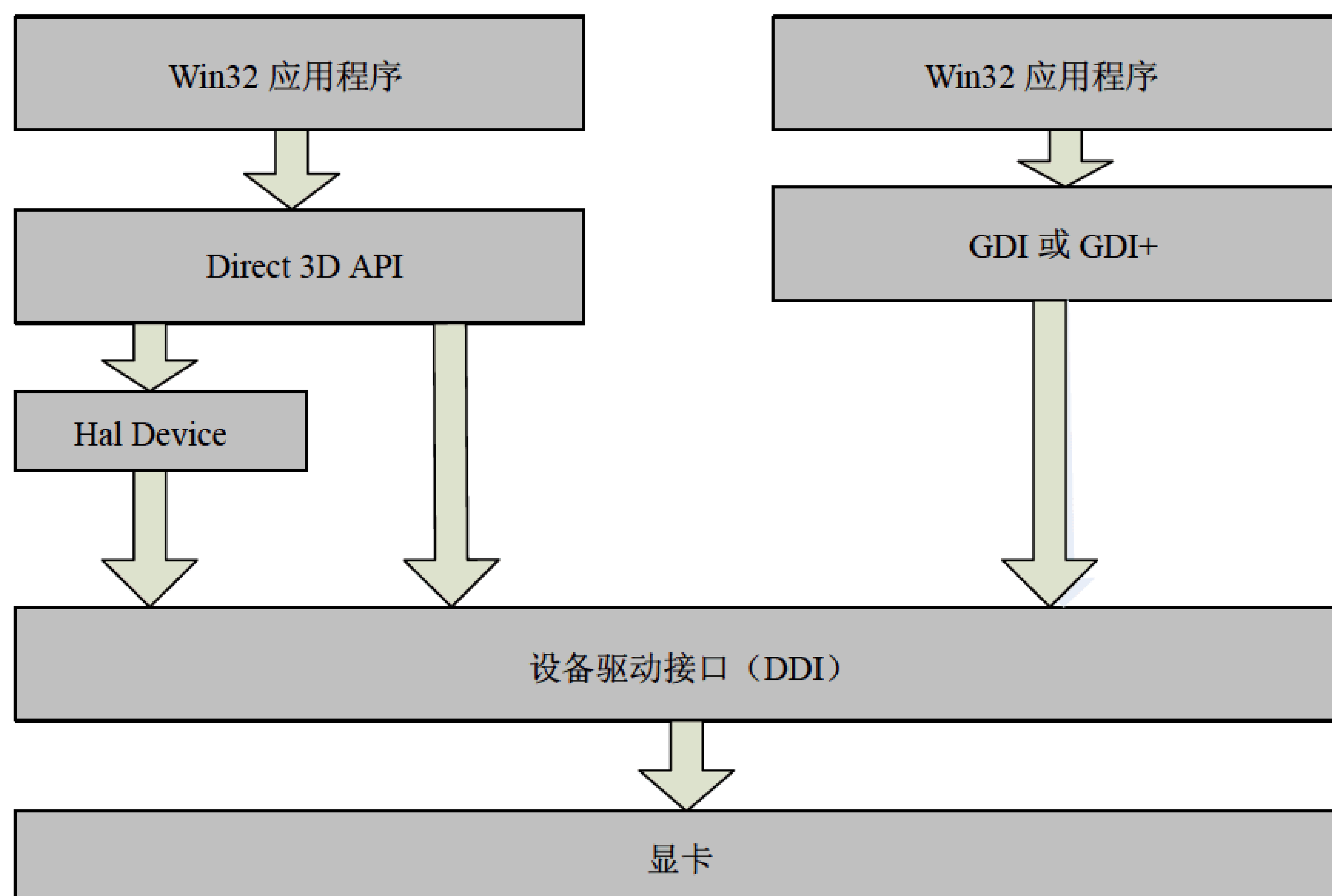


图 27-12　Direct Graphics 应用程序的程序架构

从图 27-12 可以看出，使用 Direct Graphics 组件的应用程序是调用 Direct3D API 的函

数接口完成图形的绘制，而 Direct3D API 可以像 GDI 或 GDI+一样直接访问设备驱动接口，也可以通过硬件加速来访问设备驱动接口，再由设备驱动接口来操作显卡。因此，在使用 Direct Graphics 组件操作图形时，需要熟悉和掌握 Direct3D API 的使用。本节下面的几小节以绘制基本三角形面为例，讲解如何操作 Direct3D API。

27.4.2　创建 IDirect3D9 接口对象

DirectX SDK 提供 Direct3DCreate9()函数来创建 IDirect3D9 接口对象，它是进行 DirectX3D 操作时必须使用的对象，表示整个 Direct3D 对象，使用它可以创建 Direct3D 设备。在 Direct3D 程序中，IDirect3D9 接口对象是第一个创建的对象，以及最后一个释放的对象。其函数原型为：

```
IDirect3D9 *WINAPI Direct3DCreate9(UINT SDKVersion);
```

其中，SDKVersion 参数值必须设置为 D3D_SDK_VERSION。如果创建成功，返回指向 IDirect3D9 接口的指针，否则返回 NULL 指针。

27.4.3　创建 Direct3D 设备

使用 Direct3D 对象可以枚举当前系统中可用的显示适配器，并通过其 IDirect3D9::CreateDevice()接口函数创建 Direct3D 设备对象。当枚举显示适配器时，不包含用户动态添加的显示适配器，如果要包含这些动态添加的显示适配器，需要重新创建 Direct3D 对象。使用相同的 Direct3D 对象创建的 Direct3D 设备共享相同的硬件显卡资源，因此，在一个 Direct3D 对象中创建多个 Direct3D 设备对象，会影响图形渲染的性能。创建 Direct3D 设备的过程分为以下 3 步。

（1）判断当前显卡是否支持 3D 顶点的硬件渲染，从而确定应该使用硬件抽象层 HAL 的哪种方式。IDirect3D9 接口对象提供 GetDeviceCaps()函数实现显卡的功能检测。其函数原型为：

```
HRESULT GetDeviceCaps(
  UINT Adapter,                   //指定要获取功能信息的显卡适配器的原始编号
  D3DDEVTYPE DeviceType,          //指定要获取功能信息的设备的类型
  D3DCAPS9 *pCaps);               //指定存放设备功能信息的 D3DCAPS9 结构指针
```

如果函数操作成功，返回 D3D_OK；如果函数失败，则返的值可能是如下取值。

- ❑ D3DERR_INVALIDCALL：方法调用无效，如传入的参数无效。
- ❑ D3DERR_INVALIDDEVICE：请求的设备类型无效。
- ❑ D3DERR_OUTOFVIDEOMEMORY：没有足够的显存来完成操作。

（2）通过 IDirect3D9 接口对象的 GetAdapterDisplayMode()函数获取适配器的当前显示模式。其函数原型为：

```
HRESULT GetAdapterDisplayMode(
  UINT Adapter,               //指定要获取当前显示模式的适配器的编号
  //指向 D3DDISPLAYMODE 结构用于存放获取的当前适配器模式信息的结构指针
  D3DDISPLAYMODE * pMode);
```

如果函数成功，则返回 D3D_OK；如果适配器编号超出范围或适配器显示模式无效，则返回 D3DERR_INVALIDCALL。

（3）通过 IDirect3D9 接口对象的 CreateDevice()函数创建代表显示适配器的设备，其函数原型为：

```
HRESULT CreateDevice(
  UINT Adapter,                  //指定要获取当前显示模式的适配器的编号
  D3DDEVTYPE DeviceType,         //表示要创建的设备类型
  HWND hFocusWindow,             //程序从前台切换到后台的通知对话框
  DWORD BehaviorFlags,           //指定创建设备的选项
  D3DPRESENT_PARAMETERS * pPresentationParameters,
                                 //指向 D3DPRESENT_PARAMETERS 结构的指针
  IDirect3DDevice9 ** ppReturnedDeviceInterface);
                                 //创建后返回的 IDirect3DDevice9 接口指针
```

27.4.4　创建顶点缓冲区

要绘制三角形面，在成功地创建完 IDirect3D9 接口对象和 Direct3D 设备后，接下来需要创建存储要绘制的三角形面上的顶点数据的缓冲区。主要分为如下 4 个步骤。

（1）调用 IDirect3DDevice9 设备对象的 CreateVertexBuffer()函数创建顶点缓冲区对象 IDirect3DVertexBuffer9，函数原型为：

```
HRESULT CreateVertexBuffer(
  UINT Length,                       //指定顶点缓冲区的大小，单位是字节
  DWORD Usage,                       //表示缓冲区的用途
  DWORD FVF,                         //D3DFVF 顶点格式常数的组合
  D3DPOOL Pool,                      //资源中有效的内存类型
  //指向 IDirect3DVertexBuffer9 接口类型的指针，其表示创建的顶点缓冲区资源
  IDirect3DVertexBuffer9** ppVertexBuffer,
  HANDLE* pSharedHandle);            //预留
```

其中 FVF 参数是 D3DFVF 顶点格式常数的组合，用于说明顶点缓冲区中存储的顶点数据使用的格式。读者可以使用此参数指定使用现有的顶点格式，也可以指定使用自定义格式的顶点数据。如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALID-CALL、D3DERR_OUTOFVIDEOMEMORY 或 E_OUTOFMEMORY，分别表示无效的调用、显存不足和内存溢出。

（2）调用 IDirect3DVertexBuffer9 对象的 Lock()函数，锁定指定范围的顶点数据，并返回顶点缓冲区内存的指针，其函数原型为：

```
HRESULT Lock(
  UINT OffsetToLock,                 //要锁定的顶点数据的偏移量，单位是字节
  UINT SizeToLock,                   //指定要锁定的顶点数据的大小，单位是字节
  VOID ** ppbData,                   //包含返回的顶点数据的内存缓冲区的指针
  DWORD Flags);                      //锁定缓冲区的选项
```

其中，如果要锁定整个顶点缓冲区，则将 SizeToLock 参数和 OffsetToLock 参数都设置为 0。ppbData 参数是包含返回的顶点数据的内存缓冲区的指针。Flags 参数是锁定缓冲区的选项，可以指定是否保存顶点缓冲区数据、锁定顶点缓冲区数据是否是只读的等等。

如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。

（3）使用内存数据复制函数，将顶点数据复制到锁定的顶点数据缓冲区指针指向的内存区域中。

（4）调用 IDirect3DVertexBuffer9 对象的 Unlock()函数，解锁对顶点数据缓冲区的锁定，其函数原型为：

```
HRESULT Unlock();
```

如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。

经过这 4 个过程，顶点缓冲区就创建成功了。

27.4.5　启动管道流水线进行渲染

准备工作完成后，就可以启动管道流水线进行三角形面的渲染了。其过程主要分为以下几个步骤。

（1）调用 IDirect3DDevice9 设备对象的 Clear()函数，清空元素，可以清除一个或多个表面，如可以清除单个渲染目标、多个渲染目标、单个模板缓存或深度缓存。函数原型为：

```
HRESULT Clear(
  DWORD Count,            //指定 pRects 参数中要清空元素的矩形个数
  CONST D3DRECT * pRects,//用于存储要清空元素的矩形
  DWORD Flags,            //一个或多个 D3DCLEAR 标记的组合，用于指定要清除的类型
  D3DCOLOR Color,         //指定用于清空渲染目标所使用的 ARGB 颜色值
  float Z,                //指定使用新的 Z 值来清空深度缓存，其值的范围为 0～1
  DWORD Stencil);         //指定使用新值清空模板缓存
```

如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。

（2）调用 IDirect3DDevice9 设备对象的 BeginScene()函数，启动场景的绘制。其函数原型为：

```
HRESULT BeginScene();
```

如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示已经调用过此函数，但是还没有调用 EndScene()函数结束场景绘制，无法中断上次场景绘制而开始新场景绘制。

（3）调用 IDirect3DDevice9 设备对象的 SetStreamSource()函数，将顶点缓冲区绑定到设备数据流中。其函数原型为：

```
HRESULT SetStreamSource(
  UINT StreamNumber,            //指定数据流，其范围从 0 到设备最大的流数目
  //指向 IDirect3DVertexBuffer9 接口的指针，表示要绑定到设备数据流的顶点缓冲区对象
  IDirect3DVertexBuffer9 * pStreamData,
  UINT OffsetInBytes,           //表示要绑定的顶点数据开始的偏移量，单位是字节
  UINT Stride);                 //表示顶点数据的大小
```

如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。

（4）调用 IDirect3DDevice9 设备对象的 SetFVF()函数，设置当前的顶点流中的顶点数据类型。其函数原型为：

```
HRESULT SetFVF( DWORD FVF);
```

其中，FVF 参数是表示顶点类型的 DWORD 值，是 D3DFVF 枚举值的组合。如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。

（5）调用 IDirect3DDevice9 设备对象的 DrawPrimitive()函数，绘制图元。此函数会渲染当前数据输入流中指定类型的图元。其函数原型为：

```
HRESULT DrawPrimitive(
  D3DPRIMITIVETYPE PrimitiveType,          //要渲染的图元的类型
  UINT StartVertex,                        //要载入的第一个顶点的索引
  UINT PrimitiveCount);                    //要渲染的顶点数目
```

其中，PrimitiveType 参数是 D3DPRIMITIVETYPE 枚举类型的数据，用于表示要渲染的图元的类型。其有效取值如表 27-1 所示。

表 27-1　图元类型

D3DPRIMITIVETYPE 枚举值	图 元 类 型
D3DPT_POINTLIST	将顶点作为独立的点进行渲染
D3DPT_LINELIST	将顶点作为直线段进行渲染
D3DPT_LINESTRIP	将顶点作为连接线进行渲染
D3DPT_TRIANGLELIST	将顶点作为独立的三角形进行渲染，每 3 个顶点作为一组，定义为分离的三角形
D3DPT_TRIANGLESTRIP	将顶点作为相连的三角形带进行渲染
D3DPT_TRIANGLEFAN	将顶点作为三角扇形进行渲染

其中，PrimitiveCount 参数指定要渲染的顶点数目，渲染的数目与图元的类型是相关的。例如，如果要渲染直线段，则每个图元有两个顶点。如果是三角形图元，则每个图元有 3 个顶点。如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。

（6）绘制完毕后，调用 IDirect3DDevice9 设备对象的 EndScene()函数，结束场景的绘制。它与 BeginScene 必须是成对调用的，所以此函数的调用是不能省略的。如果没有调用此函数，则下次调用 BeginScene()函数时会失败。同时，调用 Present()函数为下一次渲染图元做准备。这样，一次绘制图元的过程就完成了。

27.4.6　实例——绘制一个基本的三角形面

结合前面几小节介绍的绘制二维图元的过程，本小节以一个实例讲解如何绘制一个基本的三角形面。分为以下 3 个基本过程。

（1）创建 Direct3D 接口对象和 Direct3D 设备，并初始化要绘制的图元的顶点缓冲区。其代码如下：

```
01  //创建 Direct3D 接口对象和 Direct3D 设备
```

```
02  BOOL TRIANGLE3D::CreateD3DDevice(HWND hWnd, BOOL fullScreen)
03  {
04      //创建 Direct3D 接口对象，并返回接口
05      m_d3d = Direct3DCreate9(D3D_SDK_VERSION);
06      if(m_d3d == NULL)
07          return false;
08      //检测适配器功能
09      D3DCAPS9 d3dCaps;
10      m_d3d->GetDeviceCaps(D3DADAPTER_DEFAULT,
11          D3DDEVTYPE_HAL, &d3dCaps);
12      BOOL hp;                                    //是否支持硬件配置
13      if(d3dCaps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT)
14          hp = true;
15      else
16          hp = false;
17      //获取适配器的当前显示模式
18      D3DDISPLAYMODE display_mode;
19      if(FAILED(m_d3d->GetAdapterDisplayMode(D3DADAPTER_DEFAULT,
20          &display_mode)))
21          return false;
22      //设置 Direct3D 设备的当前参数
23      D3DPRESENT_PARAMETERS param = {0};
24      param.BackBufferWidth        = RENDER_WIDTH;
25      param.BackBufferHeight       = RENDER_HEIGHT;
26      param.BackBufferFormat       = display_mode.Format;
27      param.BackBufferCount        = 1;
28      param.hDeviceWindow          = hWnd;
29      param.Windowed               = !fullScreen;
30      param.SwapEffect             = D3DSWAPEFFECT_FLIP;
31      param.PresentationInterval   = D3DPRESENT_INTERVAL_DEFAULT;
32      //创建代表显示适配器的设备
33      DWORD flags = hp ? D3DCREATE_HARDWARE_VERTEXPROCESSING :
34          D3DCREATE_SOFTWARE_VERTEXPROCESSING;
35      HRESULT hr;
36      if(FAILED(hr=m_d3d->CreateDevice(D3DADAPTER_DEFAULT,
37          D3DDEVTYPE_HAL,hWnd, flags, &param, &m_d3dDevice)))
38          return false;
39      return true;
40  }
```

上面代码中，调用 Direct3DCreate9()函数创建 Direct3D 接口对象后，调用 GetDeviceCaps()函数检测设备的功能，调用 GetAdapterDisplayMode()函数获取显示适配器的模式，并根据需要设置创建设备的 D3DPRESENT_PARAMETERS 参数，最后调用 CreateDevice()函数创建显示适配器设备。接下来就是初始化顶点缓冲区，代码如下：

```
01  //使用自定义的顶点结构初始化顶点缓冲区
02  BOOL TRIANGLE3D::InitVertexBuffer()
03  {
04      CUSTOM_VERTEX customVertex[] =
05      {
06          { 300.0f, 100.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255, 0, 0) },
07          { 500.0f, 500.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(0, 0, 255) },
08          { 100.0f, 500.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(0, 255, 255)},
09      };
10      BYTE* vertexData;                               //创建顶点缓冲区
11      if(FAILED(m_d3dDevice->
12          CreateVertexBuffer(3 * sizeof(CUSTOM_VERTEX),0,
13          CUSTOM_VERTEX_FVF, D3DPOOL_MANAGED, &m_vertexBuffer, NULL)))
```

```
14          return false;
15      //锁定数据范围，并获取指向缓冲区内存的指针
16      if(FAILED(m_vertexBuffer->Lock(0, 0, (void**) &vertexData, 0)))
17          return false;
18      memcpy(vertexData, customVertex, sizeof(customVertex));
19      //复制顶点数据到缓冲区中
20      m_vertexBuffer->Unlock();                      //解锁顶点缓冲区
21      return true;
22  }
```

上面代码通过 Direct3D 设备的 CreateVertexBuffer()函数创建顶点缓冲区对象，并使用 Lock()方法锁定后，使用 memcpy()内存数据复制函数复制已经定义好的顶点数据，此处顶点数据类型为 CUSTOM_VERTEX，是自定义数据类型，复制完毕后，解锁对顶点缓冲区的锁定。

（2）初始化完成后，就可以进行三角形面的渲染了。代码如下：

```
01  //渲染三角形面
02  void TRIANGLE3D::Render()
03  {
04      if(m_d3dDevice == NULL)
05          return;
06      //使用黑色清除渲染目标
07      m_d3dDevice->Clear(0, NULL, D3DCLEAR_TARGET,
08          D3DCOLOR_XRGB(0, 0, 0),1.0, 0);
09      //开始渲染绘制
10      m_d3dDevice->BeginScene();
11      //绑定顶点缓冲区到设备数据流
12      m_d3dDevice->SetStreamSource(0, m_vertexBuffer,
13          0, sizeof(CUSTOM_VERTEX));
14      //设置当前顶点流的顶点格式
15      m_d3dDevice->SetFVF(CUSTOM_VERTEX_FVF);
16      //渲染三角形面
17      m_d3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
18      //结束渲染绘制
19      m_d3dDevice->EndScene();
20      //为下一次缓冲区渲染做准备
21      m_d3dDevice->Present(NULL, NULL, NULL, NULL);
22  }
```

上面代码首先调用 Clear()函数清除渲染目标中的当前元素。调用 BeginScene()函数启动场景绘制，并调用 SetStreamSource()函数和 SetFVF()函数设置顶点数据缓冲区和顶点数据类型，然后调用 DrawPrimitive()函数绘制图元。最后，调用 EndScene()函数结束场景绘制，并调用 Present()函数为下一次渲染做准备。

（3）在执行完图元的绘制后，不要忘记释放资源。代码如下：

```
01  void TRIANGLE3D::ReleaseD3D()        //释放 Direct3D 资源
02  {
03      SAFE_RELEASE(m_vertexBuffer);    //释放顶点缓冲区
04      SAFE_RELEASE(m_d3dDevice);       //释放 3D 设备
05      SAFE_RELEASE(m_d3d);
06  }
```

上面这 3 步是实现绘制三角形面类 TRIANGLE3D 的代码，在测试程序中分别调用这 3 部分中的函数即可完成绘制工作。代码如下：

```
01  BOOL CDrawTriangleDlg::OnInitDialog()
02  {
03      ...
04      if(! triangle.CreateD3DDevice(m hWnd, true))
05          return false;//创建设备
06      if(! triangle.InitVertexBuffer())
07          return false;
08      ...
09  }
```

上面代码在对话框初始化函数中创建 Direct3D 设备，并初始化顶点缓冲区。下面代码用于检测用户是否按下 D 键，如果按下 D 键，则开始渲染图元。

```
01  void CDrawTriangleDlg::OnKeyDown(UINT nChar,
02                                      UINT nRepCnt, UINT nFlags)
03  {
04      if ((nChar == 'd') || (nChar == 'D'))
05         triangle.Render();
06      //如果按下 D 键，则绘制三角形
07      CDialog::OnKeyDown(nChar, nRepCnt, nFlags);
08  }
```

这样，就完成了基本三角形面的绘制，程序运行效果图如图 27-13 所示。

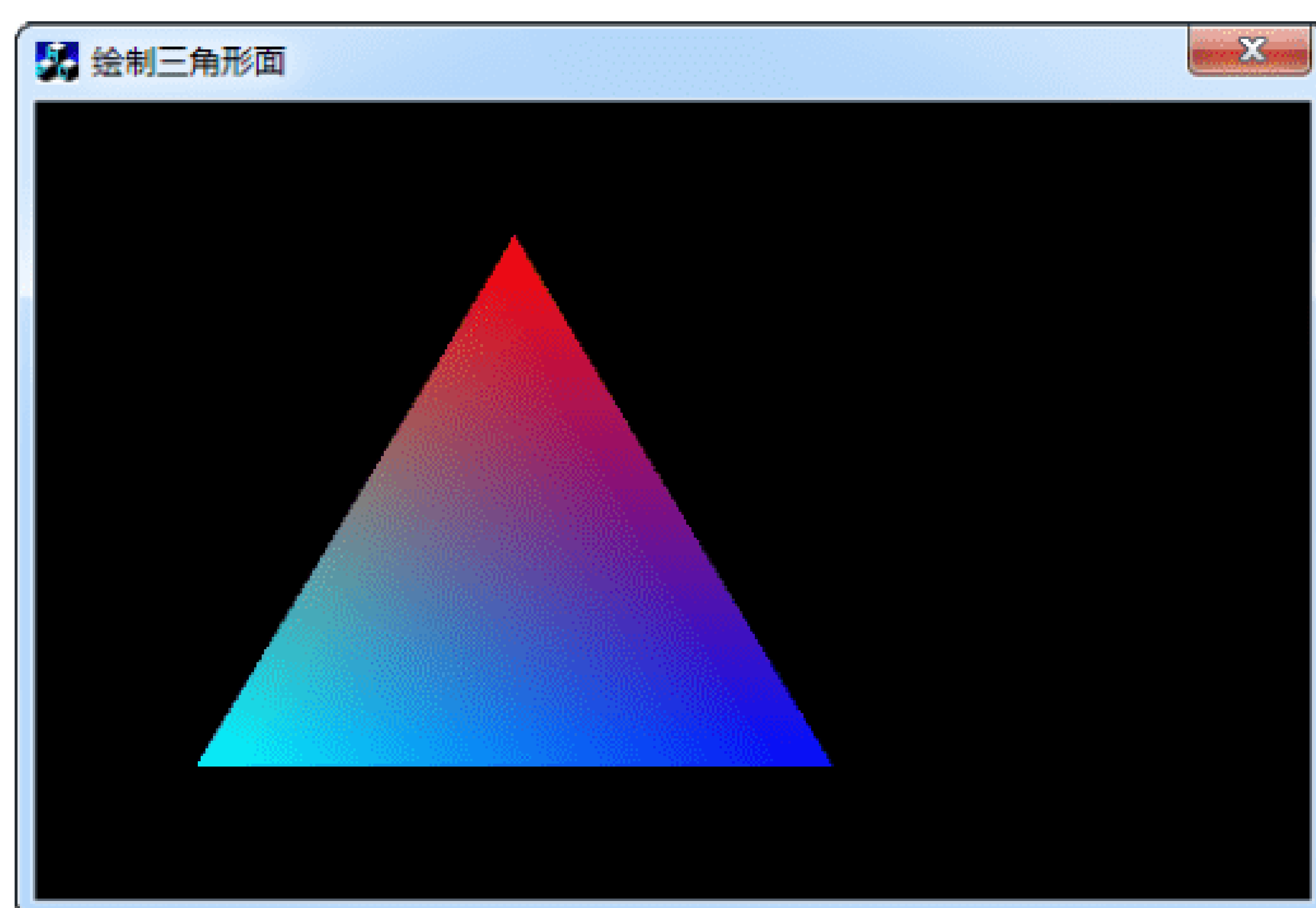


图 27-13　绘制基本三角形面运行效果

上面示例中绘制了一个三角形，3 个顶点的颜色分别为红色、绿色和蓝色，三角形中的颜色会以 3 个顶点的颜色渐变来渲染。

27.5　基本立体面的绘制

27.4 节介绍的如何使用 Direct3D 进行基本三角形面的绘制，是绘制二维图形，本节在 27.4 节的基础上，深入讲解三维立体面的绘制，着重讲述在进行立体面绘制时，需要了解的知识及方法。通过本节的讲解，读者应该能够掌握基本的三维物体的绘制方法。

27.5.1　3D 原始类型

一个 3D 图元是组成单个 3D 元素的顶点的集合。在 3D 坐标系中，最简单的图元是点

集合，称为点列表。多边形是稍微复杂一点的 3D 元素，其是由至少 3 个顶点组成的封闭区域。最简单的多边形是三角形。Direct3D 使用三角形来组成其他多边形，因为三角形的 3 个顶点保证是共面的，而要渲染非共面的顶点，效率是很低的，因此，使用三角形组合成更大的、复杂的多边形和网格。下面讲解 Direct3D 设备所支持的 3D 基本原始类型。

点列表是顶点集合，被渲染为独立点。在 3D 场景中，可以使用点作为开始点，或是多边形面上的分割线。应用程序可以对点列表应用材质和纹理，在点上绘制材质或纹理的颜色，只在点上有效，在点之间不会有任何颜色显示。以下代码显示了如何创建点列表中的顶点。

```
01  struct CUSTOMVERTEX{float x,y,z;};
02  CUSTOMVERTEX Vertices[] =
03  {
04      {-5.0, -5.0, 0.0}, { 0.0, 5.0, 0.0},
05      { 5.0, -5.0, 0.0}, {10.0, 5.0, 0.0},
06      {15.0, -5.0, 0.0},{20.0, 5.0, 0.0}
07  };
```

上面的代码定义了 6 个顶点元素，并为这 6 个顶点赋坐标值。如图 27-14 所示，显示了定义的点列表。

线列表是独立的直线段的列表，在 3D 场景中用于绘制直线。应用程序通过填充顶点数据来创建线列表。因为每条线段由两个顶点构成，因此，填充线列表的顶点数目必须是等于或大于 2 的偶数。用程序可以对线列表应用材质和纹理，在线上绘制材质或纹理的颜色，只在线上有效，在线段之间的任何地方不会受其影响。和上面列出的定义点列表代码相同的代码，如果用于定义线列表，则其定义的线列表如图 27-15 所示。

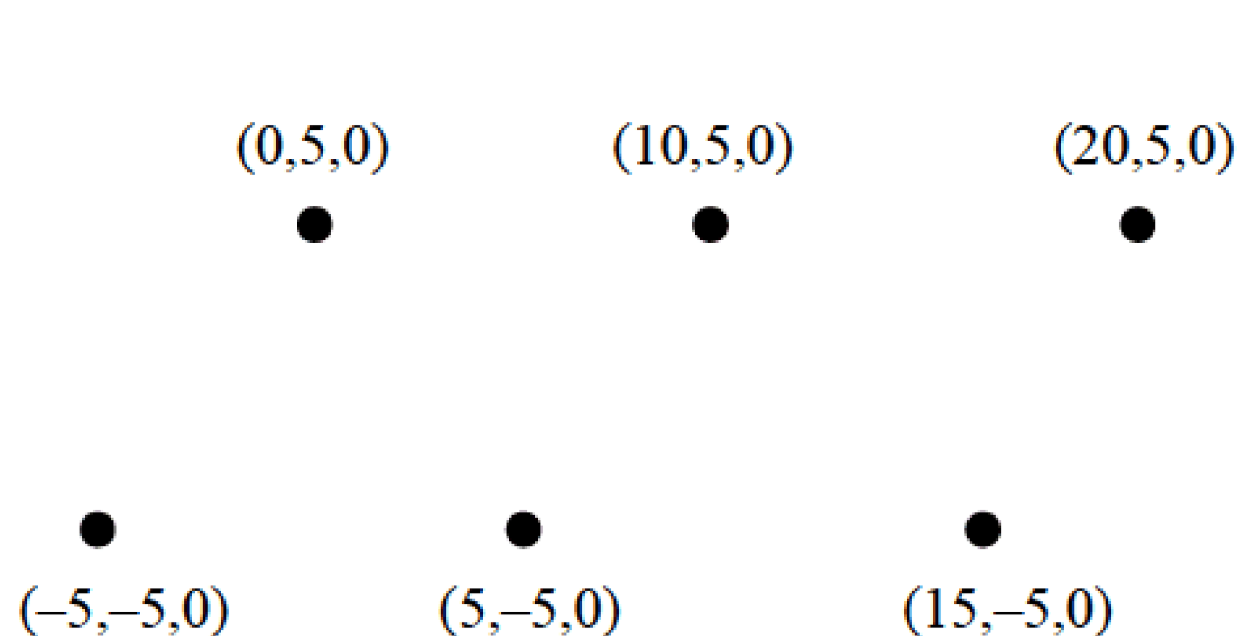


图 27-14　Direct3D 的点列表

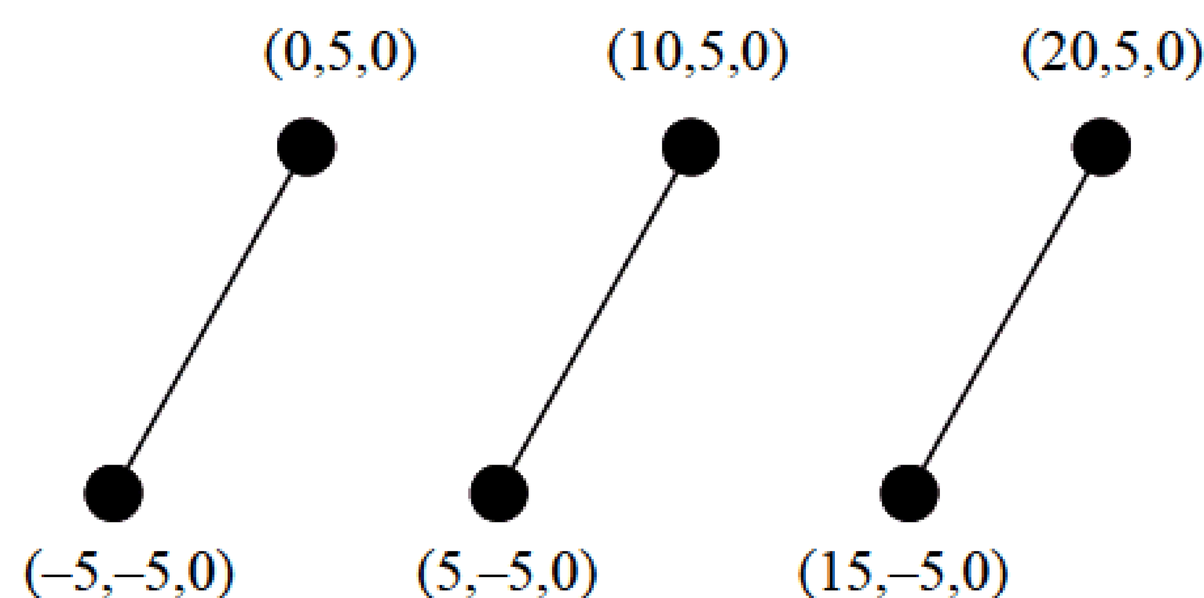


图 27-15　Direct3D 的线列表

从图 27-15 中可以看出，在定义线列表时，每两个点组成一条线段，因此点的数目必须是大于或等于 2 的偶数，否则渲染的时候会出现错误。

连接线是由连接的线段组成的图元，使用连接线可以创建不封闭的多边形，而封闭多边形是指图元的最后一个顶点与其第一个顶点使用线段连接起来。如果使用连接线组成多边形，则不能保证顶点是共面的。定义连接线列表的方法与定义点列表的方法相同，如果用于定义连接线列表，则其定义的连接线列表如图 27-16 所示。

从图 27-16 可以看出，在定义连接线列表时，会将定义的顶点依次连接起来，连接的顺序与顶点位置无关，仅与其在顶点缓冲区中的顺序有关。

三角形列表是由独立的三角形组成的图元，可能相连，也可能不相连。三角形列表至少包含 3 个顶点，顶点数目必须能够被 3 整除，会从第一个顶点开始，将连续的 3 个顶点组成一个封闭三角形。使用三角形列表可以创建由不相连的一块一块的区域组成的对象，

将要绘制的区域分割成许多小的不相连的三角形。和上面列出的定义点列表代码相同的代码，如果用于定义三角形列表，则其定义的三角形列表，如图 27-17 所示。

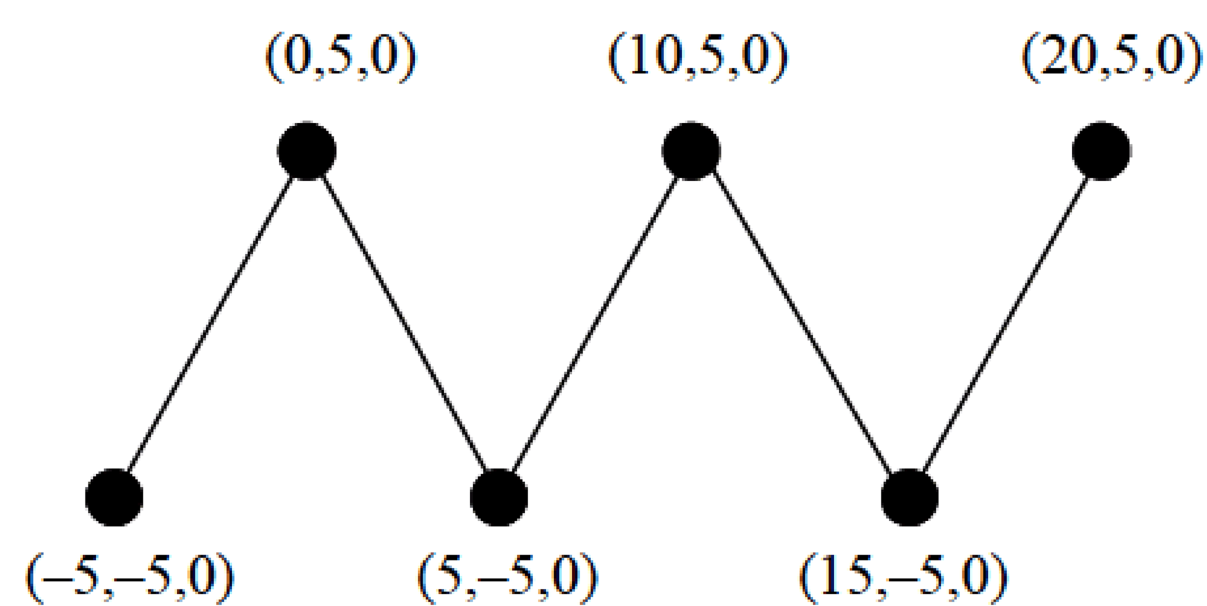


图 27-16　Direct3D 的连接线列表

从图 27-17 可以看出，在定义连接线列表时，会将定义的顶点依次连接起来，连接的顺序与顶点位置无关，仅与其在顶点缓冲区中的顺序有关。

三角形带是由相连的一系列三角形组成的图元，因为三角形是相连的，所以，应用程序不需要指定每个三角形中重复的顶点。例如和上面列出的定义点列表代码相同的代码，如果用于定义三角形带列表，则其定义的三角形带如图 27-18 所示。

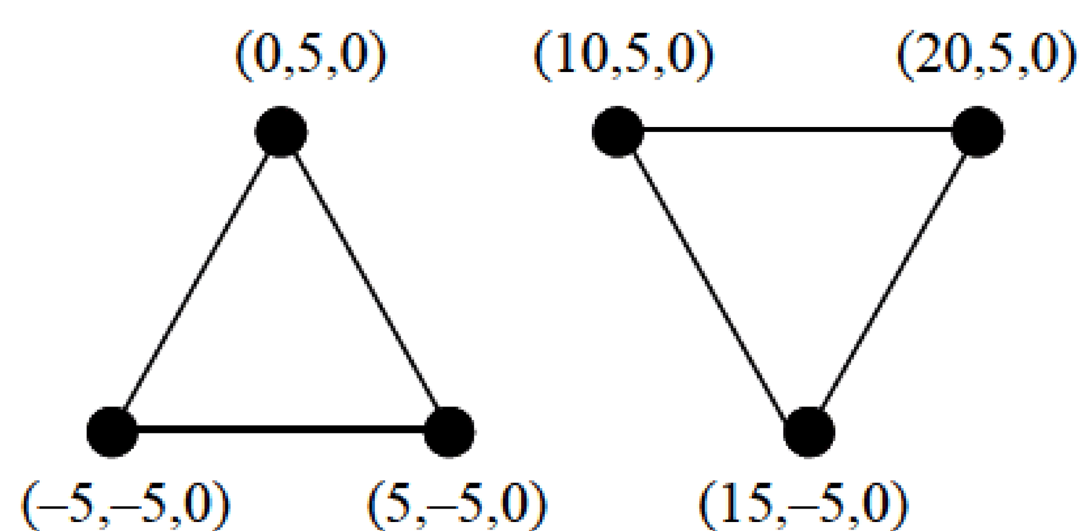


图 27-17　Direct3D 的三角形列表

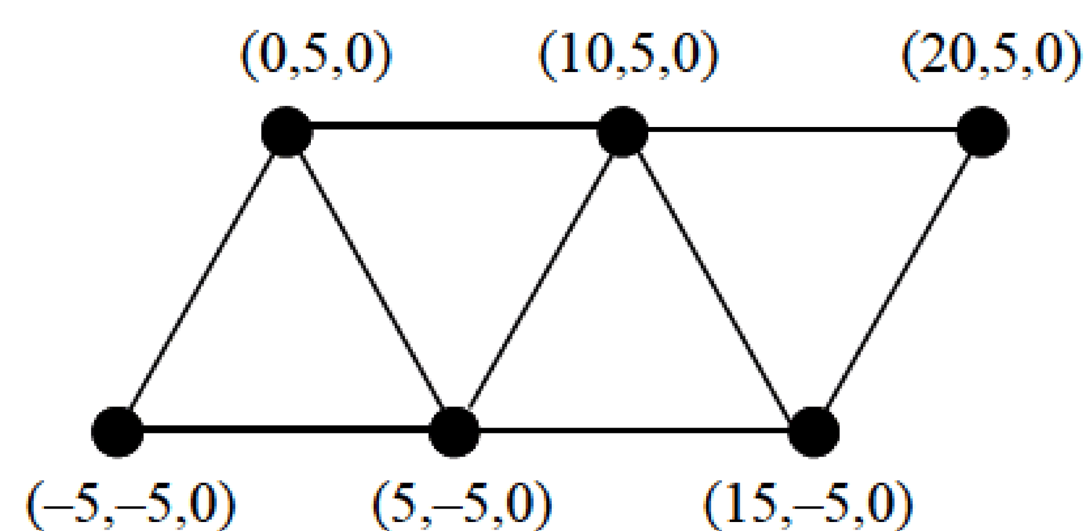


图 27-18　Direct3D 的连接三角形列表

从图 27-18 可以看出，在定义三角形带列表时，会将第一个、第二个和第三个顶点组成一个三角形，将第二个、第三个和第四个顶点组成第二个三角形，将第三个、第四个和第五个顶点组成第三个三角形，依次类推。很多 3D 场景都是由连接三角形组成的。

三角扇形列表类似于连接三角形，只是其所有三角形共用一个顶点。以下代码定义了一个三角扇形。

```
01  struct CUSTOMVERTEX{float x,y,z;};
02  CUSTOMVERTEX Vertices[] =
03  {   { 0.0, 0.0, 0.0},{-5.0, 5.0, 0.0},
04      {-4.0, 7.0, 0.0},{ 0.0, 10.0, 0.0},
05      { 4.0, 7.0, 0.0},{ 5.0, 5.0, 0.0},
06  };
```

使用 6 个顶点绘制了带有 4 个三角形的三角扇形，绘制出的效果如图 27-19 所示。

从图 27-19 可以看出，在定义三角扇形列表时，会将第一个、第二个和第三个顶点组成一个三角形，将第一、第三个和第四个顶点组成第二个三角形，将第一个、第四个和第五个顶点组成第三个三角形，依次类推。

上面介绍的这 6 种 3D 原始类型，都可以使用 Direct3D 设备的 DrawPrimitive()接口函数绘制，这在 27.4.5 小节中介绍过。

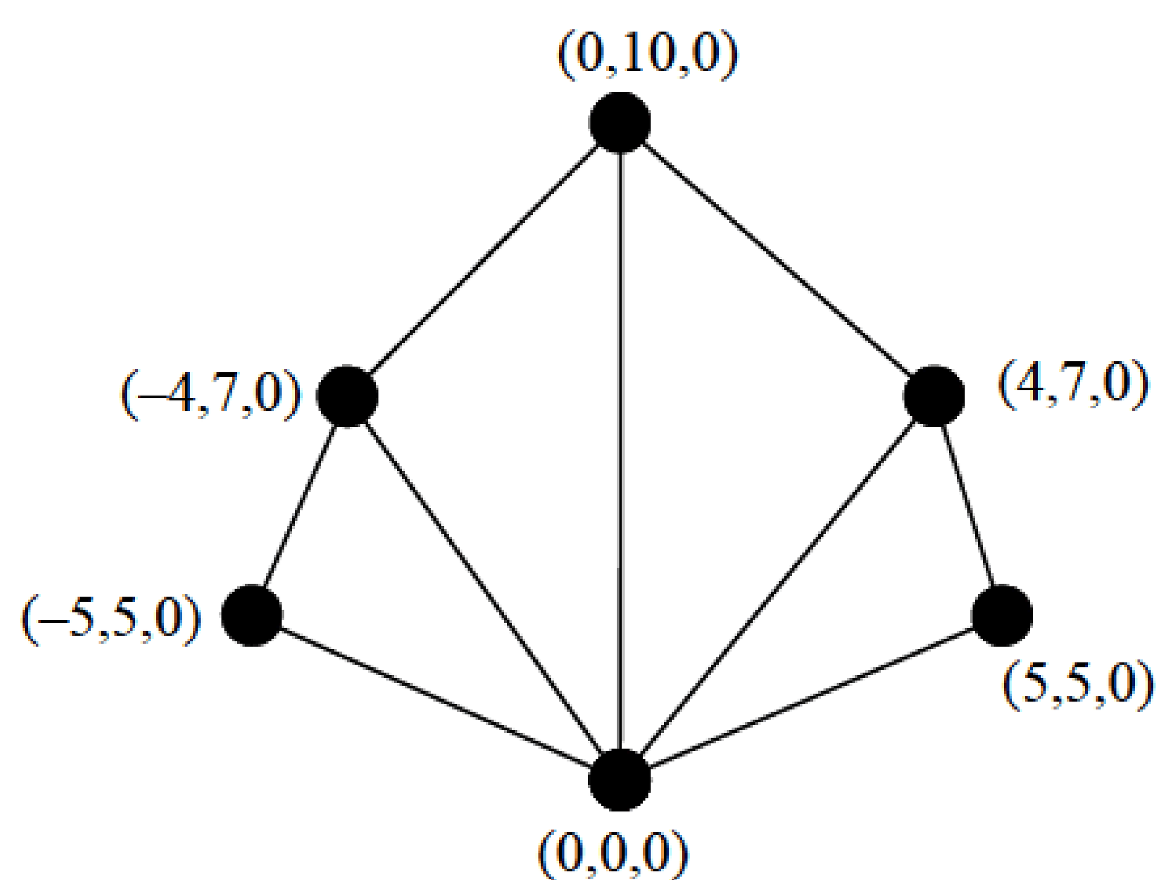


图 27-19　Direct3D 的三角扇形列表

27.5.2　背面剔除和顶点顺序

在现实世界中，面向摄像机的三维物体的剖分三角形面为前面，是可视的；背向摄像机的三维物体的剖分三角形面为背面，是不可视的。在将三维物体显示在二维屏幕上时，会渲染显示三维物体的前面，而不显示三维物体的背面，此过程称为背面剔除。

Direct3D 在渲染前会将顶点写入顶点缓冲区，写入顶点缓冲区的顺序，称为顶点顺序。在 Direct3D 中使用顶点顺序来决定指定的 3 个顶点组成的三角形是三维物体的前面还是背面。如果写入顶点缓冲区中顶点的顺序是顺时针方向，则表示此三角形是三维物体的正面，需要渲染；如果写入顶点缓冲区中顶点的顺序是逆时针方向，则表示此三角形是三维物体的背面，则不进行渲染。如图 27-20 所示，列出了顺时针顶点顺序和逆时针顶点顺序。

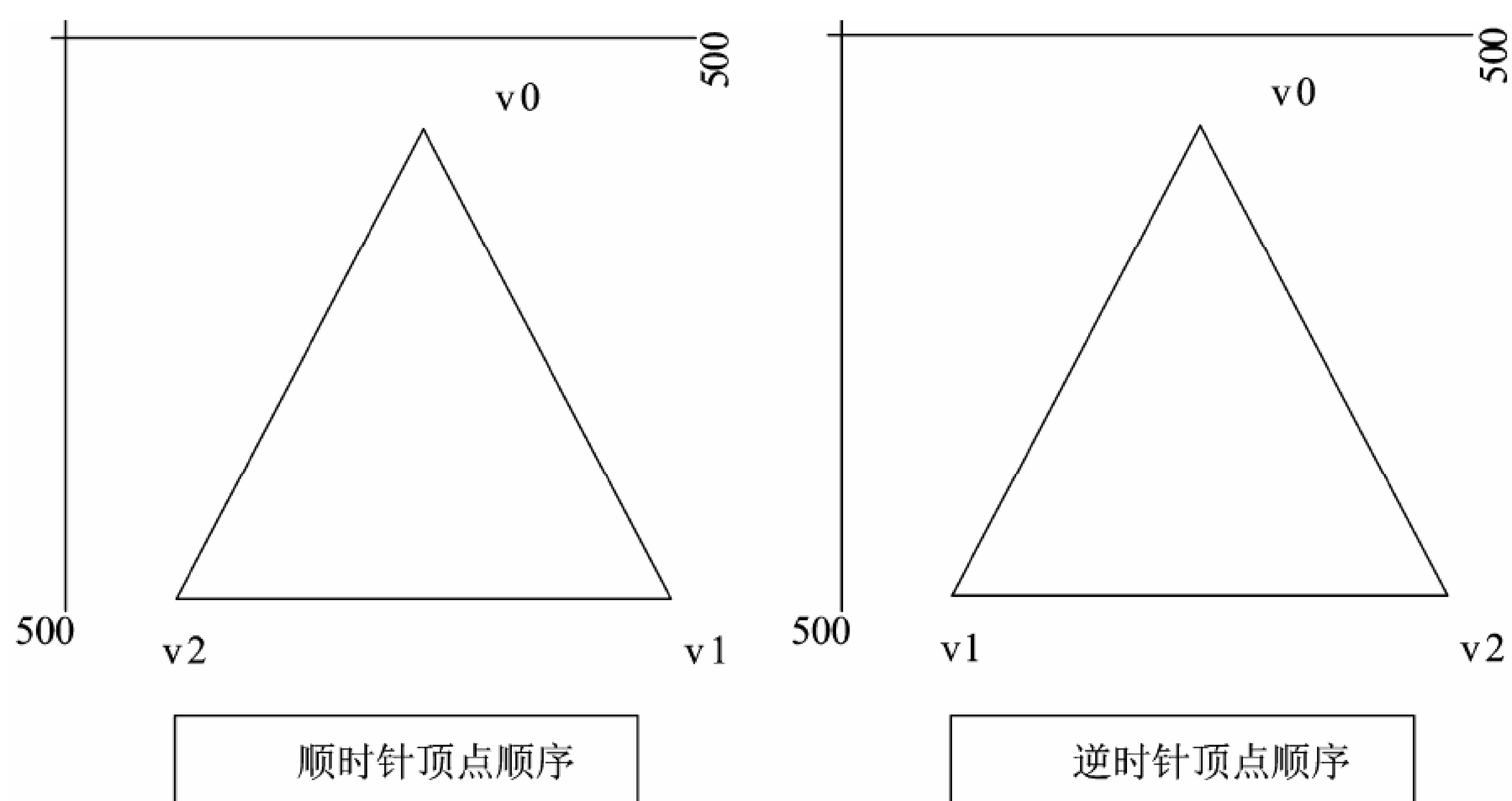


图 27-20　顺时针和逆时针顶点顺序

在图 27-20 中，顶点 V0、顶点 V1 和顶点 V2 这 3 个顶点分别组成了顺时针顶点顺序和逆时针顶点顺序，当 Direct3D 渲染三角形面时，只会渲染左边的顺时针顶点顺序指定的三角形，而不显示右边的逆时针顶点顺序指定的三角形。

27.5.3　顶点索引缓冲区

在屏幕上显示三维物体时，三维物体可以看作多个三角形面拼接而成，要渲染三维物体，则需要多次重复写入顶点数据到顶点缓冲区中。为了避免重复写入顶点数据而占用大量内存，Direct3D 提供了顶点索引缓冲区的方法。此方法会将组成渲染三维物体的所有三角形面的顶点数据的并集写入顶点缓冲区中，其中没有重复的顶点数据，然后创建一个存放每个三角形面使用的顶点的索引号的顶点索引缓冲区。在进行三维物体绘制时，通过顶点索引缓冲区中的索引数据到顶点缓冲区中读取相应的顶点数据，并进行渲染。

Direct3D 设备提供 CreateIndexBuffer()函数创建顶点索引缓冲区。其函数原型为：

```
HRESULT CreateIndexBuffer(
  UINT Length,                    //指定顶点索引缓冲区的大小，单位为字节
  DWORD Usage,                    //指定创建的顶点索引缓冲区的用途
  D3DFORMAT Format,               //顶点索引缓冲区中的数据格式，D3DFMT_INDEX16
                                  //和 D3DFMT_INDEX32
  D3DPOOL Pool,                   //描述存放顶点数据的有效的内存类型
  IDirect3DIndexBuffer9** ppIndexBuffer,//存放创建的索引缓冲区的内存地址
  HANDLE* pSharedHandle);         //预留
```

如果函数调用成功，则返回 D3D_OK；否则，返回 D3DERR_INVALIDCALL、D3DERR_OUTOFVIDEOMEMORY 和 E_OUTOFMEMORY，分别表示无效的调用、显存不足和内存溢出错误。

创建完顶点索引缓冲区后，还需要将三维物体的各个三角形面的顶点索引写入其中，在写入时，需要使用其 Lock()函数和 UnLock()函数进行锁定和解锁。

最后，还需要将三维物体的各个顶点数据写入管道流水线的顶点缓冲区中，并使用 Direct3D 对象的 SetStreamSource()函数设置其进行渲染的顶点缓冲区，使用 SetIndices()函数设置其进行渲染的顶点索引缓冲区。

由于三维物体的三角形面的顶点会有许多重复的顶点数据，使用顶点索引缓冲区可以大大减少内存的使用。因此，在进行基本立体面的绘制时，一定要使用顶点索引缓冲区机制来存储和使用三角形面的顶点数据，以提高程序性能。

27.5.4　在世界坐标系中放置物体

通常一个三维场景中，会包含多个三维物体，而每个三维物体都在其自身的局部坐标系中给出坐标点。为了将多个三维物体显示在一个场景中，则必须使用统一的世界坐标系来放置三维物体。

Direct3D 的渲染管道线提供从局部坐标到世界坐标的平移变换计算。使用 27.3.4 小节中介绍的 Direct3D 设备的 SetTransform()函数、平移变换矩阵和 D3DTS_WORLD 宏，可以设置管道流水线的世界变换矩阵。这样，当三维物体顶点的局部坐标和顶点索引值放入渲染管道流水线并开始执行管道流水线的渲染时，Direct3D 组件会自动计算三维物体顶点的世界坐标，然后在世界坐标系中统一放置物体。

27.5.5　架设摄影机进行取景和投影

计算出三维场景各个顶点的世界坐标后，需要架设虚拟摄影机，以虚拟摄影机的位置作为坐标原点、z 轴指向虚拟摄影机建立摄影坐标系，然后进行取景，确定剪裁投影的视截体区域，此区域由远近平面的位置和视域的张角来确定。取景完毕后，就可以进行三维场景的剪裁投影的处理，从而实现从三维场景到二维图像的转换。

Direct3D 的渲染管道线提供从世界坐标到摄影坐标的变换计算。使用 27.3.4 小节中介绍的 Direct3D 设备的 SetTransform()函数和摄影变换矩阵，可以设置管道流水线的摄影变换矩阵，实现摄影变换。这样，当管道流水线执行摄影变换阶段时，Direct3D 设备会根据摄影变换矩阵计算出三维物体的顶点的摄影坐标。

计算出摄影坐标之后，还需要根据取景情况，进行剪裁投影处理，将视截体范围之外的物体部分剪裁掉，并使用远小近大的原则进行剪裁投影变换。Direct3D 的渲染管道线提供剪裁投影变换计算。使用 27.3.4 小节中介绍的 Direct3D 设备的 SetTransform()函数和投影变换矩阵，可以设置管道流水线的投影变换矩阵，实现投影变换。这样，当管道流水线执行投影变换阶段时，Direct3D 设备会根据投影变换矩阵计算出二维投影坐标和远近信息。

27.5.6　屏幕视口的设置

当生成三维场景的二维投影数据后，就可以将其在计算机屏幕上对像素数据进行光栅化显示。默认情况下，使用应用程序的对话框作为输出区域，即屏幕视口，使用 27.3.4 小节中介绍的 Direct3D 设备对象提供的 SetViewport()函数可以设置视口在应用程序对话框中的位置和大小。其中的 D3DVIEWPORT9 结构定义如下：

```
typedef struct D3DVIEWPORT9 {
  DWORD X;                        //指定视口在渲染目标面上的左上角的横坐标
  DWORD Y;                        //指定视口在渲染目标面上的左上角的纵坐标
  DWORD Width;                    //指定渲染的宽度
  DWORD Height;                   //指定渲染的高度
  float MinZ;                     //指定渲染的深度范围
  float MaxZ;                     //指定渲染的深度范围
} D3DVIEWPORT9, *LPD3DVIEWPORT9;
```

使用 SetViewport()函数设置屏幕视口，在 27.3.4 小节中介绍过。

27.5.7　实例——绘制一个基本的立体面

结合前面几小节介绍的绘制三维物体的过程，本小节以一个实例讲解如何绘制一个基本的立体面——一个从上向下俯视的三角椎体。

创建 Direct3D 接口对象和 Direct3D 设备，并初始化要绘制的图元的顶点缓冲区和顶点索引缓冲区。创建 Direct3D 对象和 Direct3D 设备与绘制二维图元的方法一样，这里不再描述，其不同是，因为三维物体需要的顶点数据通常比较多，因此，最好在初始化时使用顶点索引缓冲区，其代码如下：

```
01  //使用自定义的顶点结构初始化顶点缓冲区，并建立顶点索引缓冲区
02  BOOL Cube3D::InitVertexBuffer()
03  {
04      CUSTOM_VERTEX customVertex[] =
05      {
06          {0.0, 0.0, 1.0,D3DCOLOR_XRGB(255, 255, 0)},
07          {-1.0, 0.0, 0.0,D3DCOLOR_XRGB(255, 255, 0)},
08          {0.0, 1.0, 0.0,D3DCOLOR_XRGB(255, 255, 0)},
09          {1.0, 0.0, 0.0,D3DCOLOR_XRGB(255, 255, 0)},
10          {0.0, -1.0,0.0,D3DCOLOR_XRGB(255, 255, 0)}
11      };
12      BYTE* vertexData;
13      //创建顶点缓冲区
14      if(FAILED(m_d3dDevice->CreateVertexBuffer(
15          5 * sizeof(CUSTOM_VERTEX),0,CUSTOM_VERTEX_FVF,
16          D3DPOOL_MANAGED,  &m_vertexBuffer, NULL)))
17          return false;
18      //锁定数据范围，并获取指向缓冲区内存的指针
19      if(FAILED(m_vertexBuffer->Lock(0, 0, (void**) &vertexData, 0)))
20          return false;
21      //复制顶点数据到缓冲区中
22      memcpy(vertexData, customVertex, sizeof(customVertex));
23      //解锁顶点缓冲区
24      m_vertexBuffer->Unlock();
25      //创建顶点索引缓冲区
26      if(FAILED(m_d3dDevice->CreateIndexBuffer(36 * sizeof(WORD),
27          0,D3DFMT_INDEX16, D3DPOOL_MANAGED,&m_indexBuffer, NULL)))
28          return false;
29      //锁定指定范围的索引数据，并获取索引缓冲区内存的指针
30      WORD* index_data;
31      if(FAILED(m_indexBuffer->Lock(0, 0, (void**) &index_data, 0)))
32          return false;
33      index_data[0]=0;  index_data[1]=1;  index_data[2]=2;
34      index_data[3]=0;  index_data[4]=2;  index_data[5]=3;
35      index_data[6]=0;  index_data[7]=3;  index_data[8]=4;
36      index_data[9]=0;  index_data[10]=1;  index_data[11]=4;
37      //解锁顶点索引缓冲区
38      m_indexBuffer->Unlock();
39      return true;
40  }
```

上面代码除了初始化顶点缓冲区中的数据外，还使用 Direct3D 对象的 CreateIndexBuffer()函数创建缓冲区索引对象，并将要显示的各个三角形面的顶点放入缓冲区索引对象中。因为三维物体存在系列坐标变换，其首先需要设置摄影机的位置，代码如下：

```
01  //设置摄影机位置
02  void Cube3D::SetCamera()
03  {
04      D3DXVECTOR3 eye(2.0, 1.5, -3.0);
05      D3DXVECTOR3 at(0.0, 0.0, 0.0);
06      D3DXVECTOR3 up(0.0, 1.0, 0.0);
07      D3DXMATRIX viewMatrix;
08      //生成左手视点的矩阵
09      D3DXMatrixLookAtLH(&viewMatrix, &eye, &at, &up);
10      //设置 Direct3D 设备视图转换状态
11      m_d3dDevice->SetTransform(D3DTS_VIEW, &viewMatrix);
```

```
12        D3DXMATRIX projMatrix;
13        //生成左手投影
14        D3DXMatrixPerspectiveFovLH(&projMatrix, D3DX_PI/2,800/600,1.0,
15        1000.0);
16        //设置 Direct3D 设备投影转换状态
17        m_d3dDevice->SetTransform(D3DTS_PROJECTION, &projMatrix);
18    }
```

上面代码使用 D3DXMatrixLookAtLH()函数生成左手笛卡尔坐标矩阵，并使用 SetTransform()函数设置视口转换矩阵，最后使用 D3DXMatrixPerspectiveFovLH()函数生成左手笛卡尔投影，同时使用 SetTransform()函数设置投影变换矩阵。使用 D3DXMatrixTranslation()函数可以将指定的坐标生成世界坐标，并使用 SetTransform()函数设置世界变换矩阵。代码如下：

```
01    //使用指定世界原点坐标系设置世界位置
02    void Cube3D::SetWorldPosition(float x, float y, float z)
03    {
04        D3DXMATRIX worldMatrix;
05        //使用指定偏移量构建矩阵，对象放置在世界坐标(x, y, z)处
06        D3DXMatrixTranslation(&worldMatrix, x, y, z);
07        //设置 Direct3D 设备的世界转换状态
08        m_d3dDevice->SetTransform(D3DTS_WORLD, &worldMatrix);
09    }
```

三维物体的渲染过程与二维图形的渲染过程类似，不同之处在于需要使用 SetRenderState()函数设置一些渲染状态。代码如下：

```
01    //渲染三角形面
02    void Cube3D::Render()
03    {
04        if(m_d3dDevice == NULL)
05            return;
06        m_d3dDevice->Clear(0, NULL, D3DCLEAR_TARGET,
07            D3DCOLOR_XRGB(155,0,155),1.0, 0);
08        m_d3dDevice->BeginScene();                      //开始渲染绘制
09        m_d3dDevice->SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);
10        //使用线框填充对象
11        m_d3dDevice->SetRenderState(D3DRS_LIGHTING, false);//关闭灯光
12        m_d3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
13        //关闭背面剔除
14        //绑定顶点缓冲区到设备数据流
15        m_d3dDevice->SetStreamSource(0, m_vertexBuffer, 0,
16            sizeof(CUSTOM_VERTEX));
17        m_d3dDevice->SetIndices(m_indexBuffer);//设置索引数据
18        m_d3dDevice->SetFVF(CUSTOM_VERTEX_FVF);//设置当前顶点流的顶点格式
19        D3DVIEWPORT9 fullViewport;                 //设置视口
20        m_d3dDevice->GetViewport(&fullViewport);//返回当前设备的视口参数
21        D3DVIEWPORT9 topLeftViewport;              //定义左上角的小视口
22        topLeftViewport.Width  = fullViewport.Width / 5;
23        topLeftViewport.Height = fullViewport.Height / 5;
24        topLeftViewport.X      = 0;
25        topLeftViewport.Y      = 0;
26        topLeftViewport.MinZ   = 0;
27        topLeftViewport.MaxZ   = 1;
28        m_d3dDevice->SetViewport(&topLeftViewport);//设置设备的视口
29        //根据索引，渲染立方体到顶点数组
```

```
30      m_d3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST ,
31          0, 0, 8, 0, 12);
32      m_d3dDevice->SetViewport(&fullViewport);    //设置设备的视口
33      //根据索引，渲染立方体到顶点数组
34      m_d3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
35          0, 0, 8, 0, 12);
36      m_d3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST,
37          0, 1);   //渲染三角形面
38      m_d3dDevice->EndScene();                    //结束渲染绘制
39      //为下一次缓冲区渲染做准备
40      m_d3dDevice->Present(NULL, NULL, NULL, NULL);
41  }
```

此实例在绘制二维图形的基础上，增加了切换摄影机位置的功能，通过 W、A、Z 和 D 键来控制图形向上、向左、向下和向右移动。每次移动物体后，需要根据操作修改三维物体在空间中的坐标值，在新位置处放置物体，并重新设置摄像机位置。代码如下：

```
01  void CDrawCubeDlg::OnKeyDown(UINT nChar,
02      UINT nRepCnt, UINT nFlags)
03  {
04      bCameraReset = false;
05      switch (nChar)
06      {
07      case 'a':   //左
08      case 'A':
09          bCameraReset = true;
10          bFirstRender = false;
11          cube.m_curX -= 0.2f;
12          break;
13      case 'd':   //右
14      case 'D':
15          bCameraReset = true;
16          bFirstRender = false;
17          cube.m_curX += 0.2f;
18          break;
19      case 'w':   //上
20      case 'W':
21          bCameraReset = true;
22          bFirstRender = false;
23          cube.m_curY += 0.2f;
24          break;
25      case 'z':   //下
26      case 'Z':
27          bCameraReset = true;
28          bFirstRender = false;
29          cube.m_curY -= 0.2f;
30          break;
31      default:
32          break;
33      }
34      if(bCameraReset || bFirstRender)
35      {
36          //如果需要重置摄影机或第一次渲染
37          cube.SetWorldPosition(cube.m_curX,
38              cube.m_curY, cube.m_curZ);
39          //设置世界坐标系
40          cube.SetCamera();                   //设置摄影机位置
41      }
```

```
42      cube.Render();                           //渲染基本立体面
43      CDialog::OnKeyDown(nChar, nRepCnt, nFlags);
44   }
```

程序运行效果如图 27-21 所示，当用户按下 W、A、Z 或 D 键后，可以移动三维物体的显示位置，同时左上角视口中的三维物体也会随之移动。

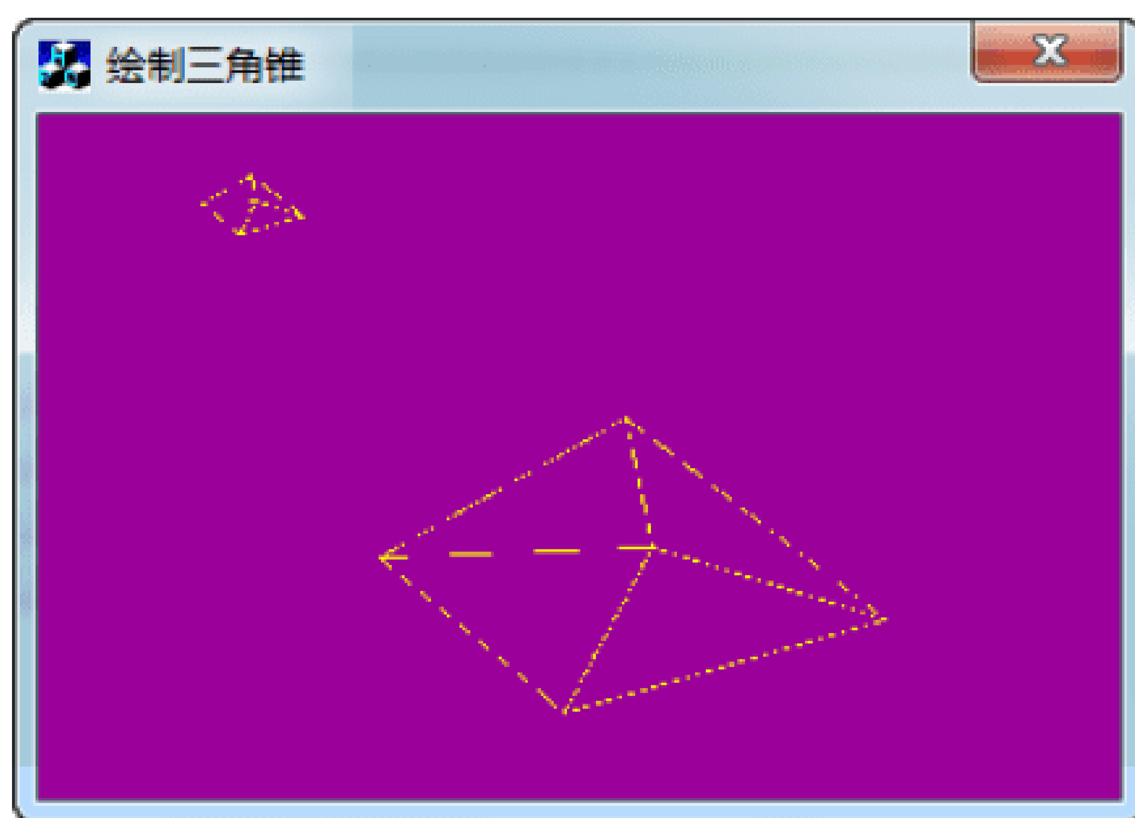


图 27-21　绘制基本立体面示例运行效果

27.6　材质和光照处理

除了前面两节中介绍的因素外，物体的材质和光照的角度对显示三维物体的效果都有影响。本节介绍显示三维物体时如何设置光源以及光源的类型，并介绍设置材质和顶点的法向量，读者可以参考 DirectX SDK 中的示例学习。

27.6.1　颜色与光照

对用户来说，颜色是在进行物体显示时最直观的因素。Direct3D 中使用 RGB 颜色法表示颜色，其中 R（Red）表示红色、G（Green）表示绿色、B（Blue）表示蓝色，使用这 3 种颜色的分量表示占有的比例，每种颜色分量值取值范围是 0～255，3 种颜色分量组合起来的颜色就是实际的颜色。其中每个分量使用一个字节来表示，因此，使用这种方法可以表示 255×255×255 种颜色。除了 RGB 外，还有个 Alpha 分量，用于表示颜色的透明度，在后面会详细讲述此分量的使用。

在 Direct3D 中的颜色使用一个 32 位的 DWORD 类型的值表示，如图 27-22 所示。

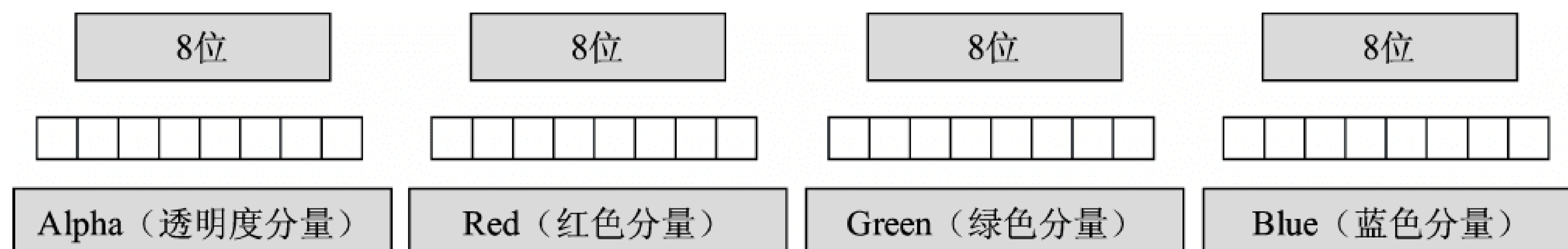


图 27-22　Direct3D 中的颜色类型

Direct3D 中使用 D3DCOLOR 来定义颜色类型，为 32 位的 DWORD 类型。其定义如下：

```
typedef DWORD D3DCOLOR;
```

与上面所讲述的一样，4 个字节分别表示组成颜色的透明度分量、红色分量、绿色分量和蓝色分量，也代表这些颜色的亮度。Direct3D 中，定义了如下与颜色有关的宏。

- D3DCOLOR_ARGB 宏：使用透明度分量、红色分量、绿色分量和蓝色分量来初始化颜色，每个分量的取值为 0～255。
- D3DCOLOR_AYUV 宏：使用透明度分量、红色亮度、蓝色亮度和绿色亮度来初始化颜色，每个分量的取值为 0～255。
- D3DCOLOR_COLORVALUE 宏：使用透明度分量、红色分量、绿色分量和蓝色分量来初始化颜色，使用每个分量占有的百分比来表示，取值为 0～1 的浮点数。
- D3DCOLOR_RGBA 宏：等同于 D3DCOLOR_ARGB，只是将透明度分量放置在最后一个参数中。
- D3DCOLOR_XRGB 宏：等同于 D3DCOLOR_ARGB，只是将透明度分量默认设置为 255，即不透明的颜色。
- D3DCOLOR_XYUV 宏：等同于 D3DCOLOR_AYUV，只是将透明度分量默认设置为 255，即不透明的颜色。

总之，颜色无论使用哪种方式表示，都是一个 32 位的整型值，4 个字节分别表示透明度分量、红色分量、绿色分量和蓝色分量。读者可以根据需要，选择合适的颜色宏进行操作。

要使得三维场景更真实，可以使用光照来加强效果，恰当地使用光照还可以增强三维物体的立体感。Direct3D 提供了对光照的支持，读者不需要指定不同光照情况下顶点的颜色，通过光照引擎和顶点原有的颜色计算光照后顶点的颜色，渲染出来的三维场景会更加逼真。

与生活中的灯光分为很多种一样，光照也分为多种，每种都有其光照模型，用以定义光照的形成和效果。每种光照通过光照资源的 3 个成员之一来照射。

- 环境光（Ambient Light）：模拟反射光，用于照亮整个场景，场景中的所有表面都会反射环境光。使用环境光照射的物体，各个部分在任何角度都被照亮。
- 漫反射（Diffuse Reflection）：是按照特殊方向进行传播的，此种灯光照射到物体表面时，会在所有方向上均匀地反射，反射光线的颜色与观察点的位置无光，是场景中的普通灯光。
- 镜面反射（Specular Reflection）：是按照特殊方向进行传播的，此种灯光照射到物体表面时，会严格按照一个方向反射，产生明亮的光泽，只能在某个角度看见。镜面灯光用在物体上产生高光的地方，如当太阳光照射在某个物体表面产生的强光，就可以使用镜面反射光来实现，此灯光的计算比环境光和漫反射光要复杂，因此 Direct3D 中默认情况下是关闭此灯光效果的，读者如果需要可以通过设置 D3DRS_SPECULARENABLE 渲染状态打开此灯光类型。

灯光是通过 D3DCOLORVALUE 或 D3DXCOLOR 来描述的，下面的代码列举了红光、蓝光和白光灯光的定义。

```
D3DXCOLOR redLight(1.0f, 0.0f, 0.0f, 1.0f);
D3DXCOLOR blueLight (0.0f, 0.0f, 1.0f, 1.0f);
D3DXCOLOR whiteLight (1.0f, 1.0f, 1.0f, 1.0f);
```

27.6.2　光源设置

在渲染三维场景时，可以通过设置光源来实现场景的光照效果。Direct3D 中使用结构 D3DLIGHT9 来定义光源，其结构定义如下：

```
typedef struct D3DLIGHT9 {
  D3DLIGHTTYPE Type;
  D3DCOLORVALUE Diffuse;        //指定灯光漫射的颜色的分量值
  D3DCOLORVALUE Specular;       //指定灯光反射镜面的颜色的分量值
  D3DCOLORVALUE Ambient;        //指定灯光的环境光颜色
  D3DVECTOR Position;           //指定在世界坐标中灯光的位置
  D3DVECTOR Direction;          //指向的世界坐标中的方向
  float Range;                  //指定光源影响的最远距离，其最大值为 FLT_MAX
  float Falloff;       //指定 Theta 成员和 Phi 成员指定的内角和外角之间减少的光亮
  float Attenuation0;           //指定光源光照程度的改变值
  float Attenuation1;           //指定光源光照程度的改变值
  float Attenuation2;           //指定光源光照程度的改变值
  float Theta;
  float Phi;
} D3DLIGHT9, *LPD3DLIGHT;
```

其中 Type 成员指定光源的类型，光源类型分为 3 种，在 27.6.3 小节～27.6.5 小节中会分别介绍。定义好光源，就可以通过 Direct3D 设备对象的 SetLight 接口函数来设置光源，其函数原型为：

```
HRESULT SetLight(
  DWORD Index,                //指定设置的光源的索引值
  CONST D3DLight9 * pLight);//指向 D3DLight9 结构的指针，其包含设置的光源的属性
```

如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。一个 Direct3D 设备对象可以设置多个光源，可以通过 LightEnable()函数来关闭或启用某个光源，达到某些光源一起作用的效果。其函数原型为：

```
HRESULT LightEnable(
  DWORD LightIndex,             //指定要开启或关闭的光源的索引值
  BOOL bEnable);                //指定要打开灯光还是关闭灯光
```

其中，bEnable 参数指定要打开灯光还是关闭灯光。如果此值为 true，则打开灯光，如果设置为 false，则关闭灯光。此函数的功能类似于现实世界中的开关。如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。

Direct3D 中支持 3 种光源——点光源、聚集光源和方向光源，下面 3 个小节将分别介绍这 3 种光源。

27.6.3　点光源

点光源是指在世界坐标系中只有一个位置点，但向所有方向都发射光的光源。图 27-23 所示是 DirectX SDK 附带的实例中的点光源实例效果，它是点光源固定在一点，向周围所有方向都发射白色光。

27.6.4　聚焦光源

聚焦光源类似于手电筒发出的光，固定在一点，但是在指定圆锥体范围内发射光。有两个角，一个是内角，是发射光的内圈的角度；另一个是外角，是发射光的外圈的角度。图 27-24 所示是 DirectX SDK 附带的实例中的聚焦光源实例效果。

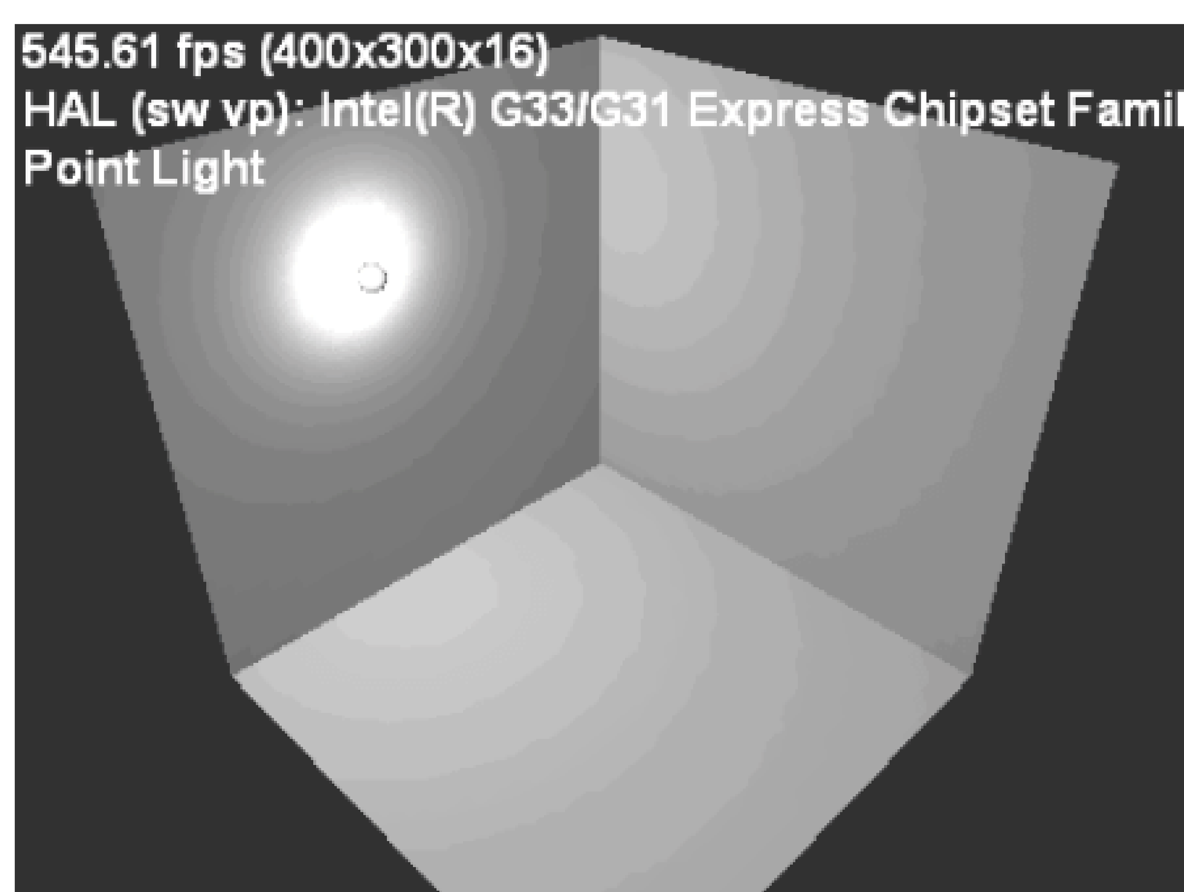


图 27-23　点光源示例

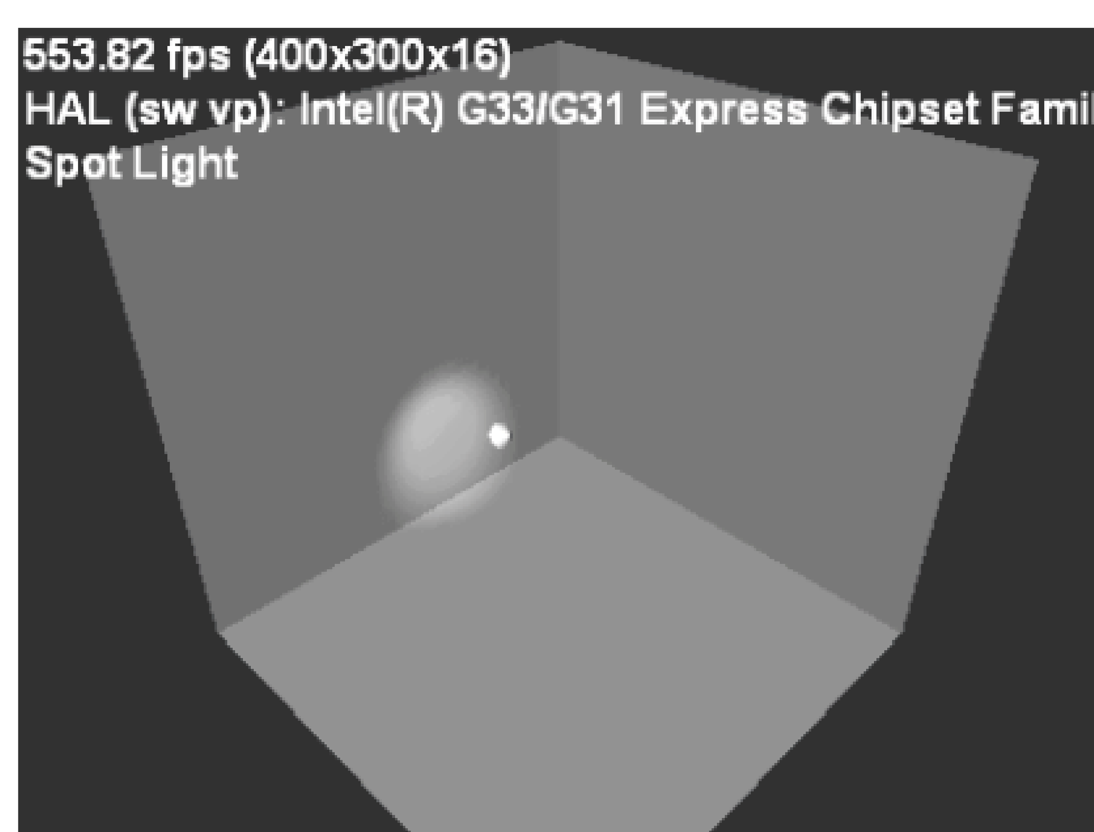


图 27-24　聚焦光源示例

图 27-24 所示是一个发出粉红色的聚焦光源的实例，在固定一点，以圆锥体的范围照射到表面上。

27.6.5　方向光源

方向光源是指向指定方向发射平行光线的光源，没有固定位置点。图 27-25 所示是 DirectX SDK 附带的实例中的方向光源实例效果。

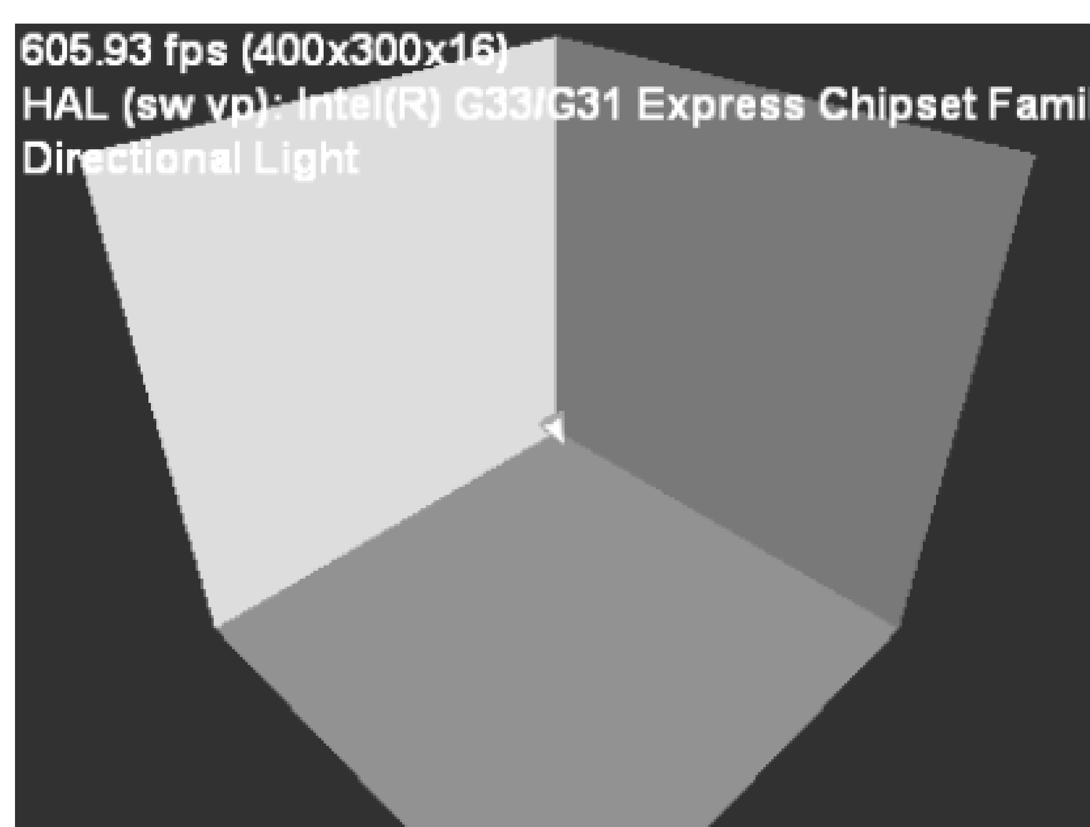


图 27-25　方向光源示例

在图 27-25 中，光源是方向光源，是绿色的照向左侧里面的光源。

27.6.6　材质设置

物体除了有颜色属性外，还有材质的不同，如金属材质的三维物体和塑料材质的物体显示出来的效果是不同的。现实世界中看到的物体颜色由物体反射回来的灯光颜色确定，如蓝色球是吸收了除蓝色以外的所有光线，蓝色光被反射回来进入人的眼睛，因此，人们

看到的球是蓝色的。不同材质反射灯光的程度是不一样的，因此，相同颜色而不同材质的物体，在人们看来颜色是有差别的。Direct3D 提供了材质来实现这种功能，即可以使用材质来定义表面反射灯光的百分比。

Direct3D 中使用 D3DMATERIAL9 结构来指定材质的属性。其结构定义为：

```
typedef struct D3DMATERIAL9 {
  D3DCOLORVALUE Diffuse;    //指定材质漫射的对应颜色分量的百分比
  D3DCOLORVALUE Ambient;    //指定材质对应颜色分量的环境颜色百分比
  D3DCOLORVALUE Specular;   //指定材质对应镜面颜色的百分比
  D3DCOLORVALUE Emissive;   //指定材质的发射颜色，使用此参数可以实现给表面添加颜
                            //色的效果，看上去就像是物体本身发出的光一样
  float Power;
} D3DMATERIAL9, *LPD3DMATERIAL9;//指定镜面的锐度，此值越大，则锐度越大
```

其中，使用 D3DRENDERSTATETYPE 可以关闭锐度效果，这样可以提高计算速度。如下代码，定义颜色是红色的材质，只反射红光，吸收其他所有颜色的光。

```
01  D3DMATERIAL9 red;
02  ::ZeroMemory(&red, sizeof(red));
03  red.Diffuse = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f);
04  red.Ambient = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f);
05  red.Specular = D3DXCOLOR(1.0f, 0.0f, 0.0f, 1.0f);
06  red.Emissive = D3DXCOLOR(0.0f, 0.0f, 0.0f, 1.0f);
07  red.Power = 5.0f;
```

上面代码设置绿色分量和蓝色分量的值为 0，表示材质不反射这两种颜色光，设置红色分量为 1，表示此材质反射 100%的红光。同样的道理，可以控制每种灯光反射的颜色，包括环境光、漫射光和镜面光。

此时，如果定义一个发出绿色光的光源，对球进行光照，则球就是黑色的。因为当前这种材质会吸收除红光以外的所有颜色的光，绿色光也被吸收，而光源又没有红色的光照在上面，因此，看起来就是黑色的。

材质需要与光照一起配合，才可以实现用户需要的效果。对于材质的使用，需要积累经验，使用的多了，才可以熟练地建立其符合要求的材质。在使用材质时，可以将其理解为现实世界中的材料，不同材料对光的反射是不同的。

在 Direct3D 中对三维场景进行渲染时，每个顶点结构是没有材质属性的，需要使用 SetMaterial()函数为 Direct3D 设备对象设置当前场景使用的材质，其函数原型为：

```
HRESULT SetMaterial( CONST D3DMATERIAL9 * pMaterial);
```

其中，pMaterial 参数是指向材质结构 D3DMATERIAL9 的指针，用于描述要设置的材质的属性。如果函数成功，则返回 D3D_OK；否则，可能返回 D3DERR_INVALIDCALL，表示无效的调用。

27.6.7　顶点的法向量

Direct3D 根据顶点法向量可以确定灯光照射到物体表面时的角度以及方向。法向量是带有方向的法线，分为两种，一种是面法向量，另一种是顶点法向量。面法向量是垂直于多边形表面的带方向的法线；顶点法向量是垂直于顶点三角形面的带方向的法线。

在渲染三维物体时，会将物体表面分解为多个三角形面，因此，在计算三角形面的法线时，需要计算每个顶点的法线，而对于三角形面，使用其面法线作为顶点法线。计算三角形面的法线步骤如下。

（1）假设构成三角形的顶点为 p0、p1、p2，其对应的顶点法线为 n0、n1、n2，计算三角形面上的两个向量：

```
u = p1 - p0
v = p2 - p0
```

（2）计算面的法向量为 n=u×v。

（3）三角形面的每个顶点法向量与面法向量是相同的，即 n0=n1=n2=n。

（4）当三角形近似表示曲面时，则使用面法线会有误差，绘制出来的曲面会出现不平滑的情况，此时需要计算顶点的平均法线。计算方法如下：

```
vn = (1/3)×(n0 + n1 + n2)
```

（5）使用 Direct3D 设备对象的 SetRenderState 方法可以设置设备的渲染状态是否初始化顶点法线，以下代码会在进行图形渲染前初始化顶点法线。

```
m_d3dDevice->SetRenderState(D3DRS_NORMALIZENORMALS, true);
```

上面的过程就是计算顶点法向量的过程，使用顶点法向量绘制的三维物体更加平滑，效果更加逼真，但是使用法线，势必会增加系统资源开销，读者应该在效果和性能之间取一个平衡设置。

27.7　纹 理 贴 图

纹理是指三维物体上的图案，像立体面的绘制方法一样，可以将一张张二维图像贴到三维物体的每个三角形的表面。当整个三维物体的所有面都贴上这些二维图像时，就形成了整个三维物体的图案纹理，此过程称为纹理贴图。纹理贴图也是物体表面的一种属性，与光照技术和物体材质结合，对三维场景进行渲染，可以使三维场景更加逼真。本节将介绍有关纹理贴图的知识，在学习本节内容时，读者可以参考 DirectXSDK 中的示例。

27.7.1　顶点的纹理坐标

纹理贴图与前面介绍的立体面的绘制相同，是对多个剖分的三角形面进行贴图来实现的。在对每个剖分三角形面进行贴图时，需要为每个顶点添加纹理坐标，确定每个三角形面的纹理图案对应在纹理图案上的纹理坐标。

纹理图像与普通图像一样，是由一系列像素点组成的，而顶点的纹理坐标就是其对应于纹理图像中相应颜色像素点的坐标。指定 3 个顶点的纹理坐标后，三角形内部的像素点的颜色值会根据顶点的颜色值进行差值计算，最后将顶点的纹理坐标和顶点坐标传递给渲染管道流水线，管道流水线就会在相应顶点处使用顶点纹理坐标和纹理图像对其进行纹理贴图。

纹理坐标一般使用 u 坐标和 v 坐标表示，其范围是 0～1，与图像的大小无关，这点尤其要注意，超过 1 的纹理坐标在纹理图像上是不存在的。至于根据纹理坐标如何获取纹理图像上相应坐标的像素的颜色值的问题，就是纹理地址模式，这在 27.7.4 小节中会介绍。

27.7.2　创建纹理对象

除了将顶点的纹理坐标和顶点坐标传给渲染管道流水线外，管道流水线要对物体进行纹理贴图，还需要将纹理图像的数据传递给管道流水线，这样管道流水线才可以从纹理图像数据中读取相应纹理坐标处的颜色值，在顶点坐标处进行渲染。Direct3D 中使用纹理对象来表示纹理图像的数据，Direct3D 设备对象提供 D3DXCreateTextureFromFile()接口函数，实现从图像文件中创建纹理对象的方法。其函数原型为：

```
HRESULT D3DXCreateTextureFromFile(
  LPDIRECT3DDEVICE9 pDevice,//指向 Direct3D 设备对象的指针
  LPCTSTR pSrcFile,          //指定纹理图案的文件名
                         //指向创建的纹理对象的 IDirect3DTexture9 接口的指针
  LPDIRECT3DTEXTURE9 * ppTexture);
```

如果函数调用成功，则返回 D3D_OK；否则，返回 D3DERR_INVALIDCALL、D3DERR_OUTOFVIDEOMEMORY 或 E_OUTOFMEMORY，分别表示无效的调用、显存不足和内存溢出错误。

除此之外，Direct3D 设备对象还提供了 D3DXCreateTextureFromFileEx()接口函数用于创建纹理对象，其函数原型为：

```
HRESULT D3DXCreateTextureFromFileEx(
  LPDIRECT3DDEVICE9 pDevice,
  LPCTSTR pSrcFile,
  UINT Width,                       //指定创建的纹理对象的宽，单位是像素
  UINT Height,                      //指定创建的纹理对象的高，单位是像素
  UINT MipLevels,
  DWORD Usage,                      //指定此纹理对象的用途
  D3DFORMAT Format,                 //指定纹理的像素格式类型
  D3DPOOL Pool,                     //描述放置纹理对象的内存类
  DWORD Filter,                     //控制图像的过滤方式
  DWORD MipFilter,                  //控制图像的过滤方式
  D3DCOLOR ColorKey,                //指定用于替换透明的黑色的颜色值
  D3DXIMAGE_INFO * pSrcInfo,        //描述源图像文件的数据信息
  PALETTEENTRY * pPalette,          //表示 256 色的调色板
  LPDIRECT3DTEXTURE9 * ppTexture);
```

使用 Direct3D 对象的 CreateTexture()函数也可以创建纹理对象。其函数原型为：

```
HRESULT CreateTexture(
  UINT Width,
  UINT Height,
  UINT Levels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  IDirect3DTexture9** ppTexture,
  HANDLE* pSharedHandle);
```

此函数的参数和返回值与 D3DXCreateTextureFromFileEx()函数的参数和返回值是相同的。

无论使用上面介绍的 3 种方法中的哪种方法创建完纹理对象后，都可以通过 IDirect3DTexture9 纹理对象的 LockRect()函数锁定指定矩形范围内的纹理数据，其函数原型为：

```
HRESULT LockRect(
  UINT Level,                                //指定要锁定的纹理资源的采样级别
  D3DLOCKED_RECT * pLockedRect,
  //指向 D3DLOCKED_RECT 结构的指针，用于描述锁定的区域
  CONST RECT * pRect,                        //指向锁定的矩形的指针
  DWORD Flags);                              //描述锁定的选项
```

如果函数调用成功，则返回 D3D_OK；否则，返回 D3DERR_INVALIDCAL，表示调用无效。此时，就可以使用纹理对象中的纹理数据了，在使用完成后，需要调用 IDirect3DTexture9 纹理对象的 UnlockRect()函数对其进行解锁。

```
HRESULT UnlockRect( UINT Level);             //指定要解锁的纹理资源的采样级别
```

如果函数调用成功，则返回 D3D_OK；否则，返回 D3DERR_INVALIDCAL，表示调用无效。Direct3D 对象提供 SetTexture()函数进行纹理设置，并且在将纹理对象传递给渲染管道流水线时，需要提供采样器的序号和纹理状态序号，以区分不同的纹理处理。

```
HRESULT SetTexture(
  D3DXHANDLE hParameter,                     //指定纹理序号
  LPDIRECT3DBASETEXTURE9 pTexture);          //指定纹理对象
```

Direct3D 对象提供 GetLevelDesc()函数，可以获取指定采样级别的纹理描述信息。其函数原型为：

```
HRESULT GetLevelDesc(
  UINT Level,                                //指定要获取的描述信息的采样级别
  D3DSURFACE_DESC * pDesc);                  //存放返回的描述信息
```

其中 pDesc 参数是指向 D3DSURFACE_DESC 结构的指针，用于存放返回的描述信息，此结构的定位如下：

```
typedef struct D3DSURFACE_DESC {
  D3DFORMAT Format;                        //表示纹理格式
  D3DRESOURCETYPE Type;                    //纹理资源类型
  DWORD Usage;                             //纹理的用法
  D3DPOOL Pool;                            //指定存放纹理数据的内存类型
  D3DMULTISAMPLE_TYPE MultiSampleType;     //指定纹理支持的全景多样本采样级别
  DWORD MultiSampleQuality;                //指定采样质量级别
  UINT Width;                              //指定纹理的宽
  UINT Height;                             //指定纹理的高
  D3DSURFACE_DESC, *LPD3DSURFACE_DESC;}
```

以上就是创建纹理对象的过程。

27.7.3　纹理过滤技术

创建了纹理对象，并为 Direct3D 设备对象设置了纹理对象，指定三角形面的顶点纹理像素点也确定了，但是还需要计算如何设置三角形面内部像素点的颜色值。一般使用差值的方法计算内部像素点的颜色值，即纹理过滤技术。常见的纹理过滤技术主要包括 3 种：Mipmap 过滤技术、MagFilter 过滤技术和 MinFilter 过滤技术。

Mipmap 过滤技术是指使用纹理图像相同但是大小不同的纹理图像，对远近不同的物体采用不同大小的纹理图像进行贴图，较远的物体采用尺寸较大的纹理图像进行贴图，较近的物体采用尺寸较小的纹理图像进行贴图，这样渲染出的三维物体的纹理效果不会有太大的失真，同时纹理贴图的速度也比较快。

MagFilter 过滤技术与 MinFilter 过滤技术，是对纹理图像进行放大或缩小到与要贴图的二维的三角形面的大小相同后，来确定三角形内部像素点的颜色值。

当使用 Direct3D 对象的 D3DXCreateTextureFromFile()函数创建纹理对象时，函数会根据纹理图像的尺寸，自动创建出一系列尺寸的纹理对象。当将纹理图像贴到三维物体表面时，会根据渲染表面的大小，自动选择合适尺寸的纹理对象进行贴图。这样可以减少三角形内部像素点颜色值的插值计算量，从而提高纹理贴图的速度。

Direct3D 设备对象提供 SetSampleState()接口函数设置渲染管道流水线使用的过滤技术类型。

```
HRESULT SetSamplerState(
  DWORD Sampler,                //指定采样阶段序号
  D3DSAMPLERSTATETYPE Type,     //指定纹理采样类型，可以定义纹理采样操作方式
  DWORD Value);                 //设置纹理采样阶段值
```

其中 Value 参数的有效取值有：

```
typedef enum D3DTEXTUREFILTERTYPE{
  D3DTEXF_NONE = 0, //表示关闭 Mipmap 过滤，而光栅贴图会使用 MagFilter 过滤技术
  D3DTEXF_POINT = 1,//点过滤使用 MagFilter 过滤技术或 MinFilter 过滤技术
  D3DTEXF_LINEAR = 2,//表示线过滤使用 MagFilter 过滤技术或 MinFilter 过滤技术
  D3DTEXF_ANISOTROPIC = 3,
  //表示各向异性纹理过滤使用 MagFilter 过滤技术或 MinFilter 过滤技术
  D3DTEXF_PYRAMIDALQUAD = 6,
  D3DTEXF_GAUSSIANQUAD = 7,
  //MagFilter 过滤技术或 MinFilter 过滤技术使用 4 采样高斯过滤
  D3DTEXF_FORCE_DWORD = 0x7fffffff,
} D3DTEXTUREFILTERTYPE, *LPD3DTEXTUREFILTERTYPE;
```

选择合适的纹理过滤技术既可以提高纹理贴图的逼真性，又可以提高纹理贴图的速度，因此要编写高效、优质的图形处理程序，需要处理好纹理过滤技术。

27.7.4　纹理地址模式

前面讲过纹理坐标[u, v]中 u 和 v 的值是在 0～1 之间的。如果顶点纹理坐标的 u 或 v 的值大于 1 或小于 0，则对应纹理图像上的像素点是不存在，因此需要使用相应的纹理地

址来寻址，确定指定纹理坐标顶点的颜色值。

纹理地址模式包括包装模式（Wrap）、镜像模式（Mirror）、夹子模式（Clamp）、边界模式（Border）和一次镜像模式（MirrorOnce）。在 Direct3D 中的定义如下：

```
typedef enum D3DTEXTUREADDRESS{
  //包装模式，此种方式只在每个整型连接点处设置纹理
  D3DTADDRESS_WRAP = 1,
  D3DTADDRESS_MIRROR = 2,
  //镜像模式，与 D3DTADDRESS_WRAP 方式类似，只是在每个整型连接点处翻转纹理
  D3DTADDRESS_CLAMP = 3,
  //夹子模式，纹理坐标超过[0,1]范围，则分别设置纹理颜色为 0 或 1
  D3DTADDRESS_BORDER = 4,
  //边界模式，纹理坐标超过[0,1]范围，则设置纹理颜色为边界颜色
  //一次镜像模式，取纹理坐标的绝对值
  D3DTADDRESS_MIRRORONCE = 5,
  D3DTADDRESS_FORCE_DWORD = 0x7fffffff,
} D3DTEXTUREADDRESS, *LPD3DTEXTUREADDRESS;
```

相同的物体，使用上面这些不同的纹理地址模式，得到的效果也不同。设置纹理地址模式的方法与设置纹理过滤技术的方法一样。关于 SetSamplerState()函数的使用，请参见 27.7.3 小节。

27.8　Alpha 颜色混合

因为影响三维物体的显示因素有多种，而这些影响最终都是通过渲染的像素颜色来影响最终效果，因此，当这些因素组合在一起时，存在一个颜色混合的问题。本节就介绍颜色混合的知识，讲解如何实现 Alpha 颜色混合。

27.8.1　颜色混合原理

在使用 Direct3D 渲染三维物体时，常常需要将顶点颜色、纹理像素颜色、光照颜色和材质反射的光颜色进行混合，显示在计算机屏幕上。在混合这些颜色时，必须设置各种颜色所占的比例，比例值由 Alpha 因子确定，使用 Alpha 颜色混合可以生成背景透明的效果，因此 Alpha 也称为透明度。

颜色混合使用混合因子来计算，假定图像的顶点颜色值为 SrcColor，纹理坐标处的像素颜色值为 DestColor，SrcBlend 为顶点颜色混合因子，DestBlend 为纹理坐标颜色混合因子，则屏幕目标颜色值为：

```
Color = SrcColor*SrcBlend + DestColor*DestBlend
```

从这个换算公式可以看出，SrcBlend 和 DestBlend 是顶点颜色和纹理颜色所占的比重。SrcColor、SrcBlend、DestColor 和 DestBlend 都是四维向量，因此，其乘法运算是向量点积运算。即：

```
SrcColor = （Sr, Sg, Sb, Sa）;
SrcBlend = （S1, S2, S3, S4）;
```

```
DestColor = （Dr, Dg, Db, Da）;
DestBlend = （D1, D2, D3, D4）;
Color = SrcColor * SrcBlend + DestColor * DestBlend;
= (Sr*S1 + Dr*D1, Sg*S2 + Dg*D2, Sb*S3 + Db*D3, Sa*S4 + Da*D4);
```

使用 Direct3D 设备对象的 SetRenderState()接口函数可以设置渲染管道流水线使用的混合因子，此时函数的第一个参数为 D3DRS_SRCBLEND 或 D3DRS_DESTBLEND，分别表示设置源混合因子和目标混合因子，代码如下：

```
m_d3dDevice.SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ZERO);
m_d3dDevice.SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```

其中，D3DBLEND_ZERO 宏和 D3DBLEND_ONE 宏是 Direct3D 中预定义的混合因子宏。Direct3D 中预定义了部分混合因子宏，其定义如下：

```
typedef enum D3DBLEND{
    D3DBLEND_ZERO = 1,            //(0, 0, 0, 0)
    D3DBLEND_ONE = 2,             //(1, 1, 1, 1)
    D3DBLEND_SRCCOLOR = 3,        //(Rs, Gs, Bs, As)
    D3DBLEND_INVSRCCOLOR = 4,     //(1-Rs, 1-Gs, 1-Bs, 1-As)
    D3DBLEND_SRCALPHA = 5,        //(As, As, As, As)
    D3DBLEND_INVSRCALPHA = 6,     //(1-As, 1-As, 1-As, 1-As)
    D3DBLEND_DESTALPHA = 7,       //(Ad, Ad, Ad, Ad)
    D3DBLEND_INVDESTALPHA = 8,    //(1-Ad, 1-Ad, 1-Ad, 1-Ad)
    D3DBLEND_DESTCOLOR = 9,       //(Rd, Gd, Bd, Ad)
    D3DBLEND_INVDESTCOLOR = 10, //(1-Rd, 1-Gd, 1-Bd, 1-Ad)
    D3DBLEND_SRCALPHASAT = 11,    //(f,f,f, 1) f = min(As, 1-Ad)
    D3DBLEND_BOTHSRCALPHA = 12, //源因子
    D3DBLEND_BOTHINVSRCALPHA = 13,
    //源因子(1-As,1-As,1-As,1-As)混合因子(As, As, As, As)
    D3DBLEND_BLENDFACTOR = 14,
    D3DBLEND_INVBLENDFACTOR = 15,
    D3DBLEND_FORCE_DWORD = 0x7fffffff,
} D3DBLEND, *LPD3DBLEND;
```

除了使用上面定义的预定义混合因子之外，读者可以根据自己的需求，自定义混合因子。但是渲染管道流水线默认是关闭 Alpha 颜色混合功能的，要使用 Alpha 颜色混合，必须调用 SetRenderState()函数设置 D3DRS_ALPHABLENDENABLE 属性为 true，打开 Alpha 颜色混合功能。代码如下：

```
m_d3dDevice.SetRenderState(D3DRS_ALPHABLENDENABLE, true);
```

27.8.2　Alpha 颜色混合例子

Alpha 颜色混合的典型例子是将一幅图像透明地显示在另一幅图像上。为了实现颜色透明的效果，Direct3D 提供了 ID3DXSprite 接口来实现。要实现颜色透明，首先需要生成 Alpha 的通道图，Direct3D 提供 DxTex.exe 工具来生成纹理图像的 Alpha 通道图，步骤如下。

（1）双击 Direct3D 安装目录下 D:\DXSDK\bin\DXUtils 中的 DxTex.exe 程序。

（2）选择 File|Open 命令，打开要绘制的前景图，此处打开 Bubules.bmp 图片。

（3）选择 Format|Change Surface Format 命令，打开 Changes Surface Format 对话框，设

置 bmp 图像的像素颜色格式为 A8R8G8B8，如图 27-26 所示。

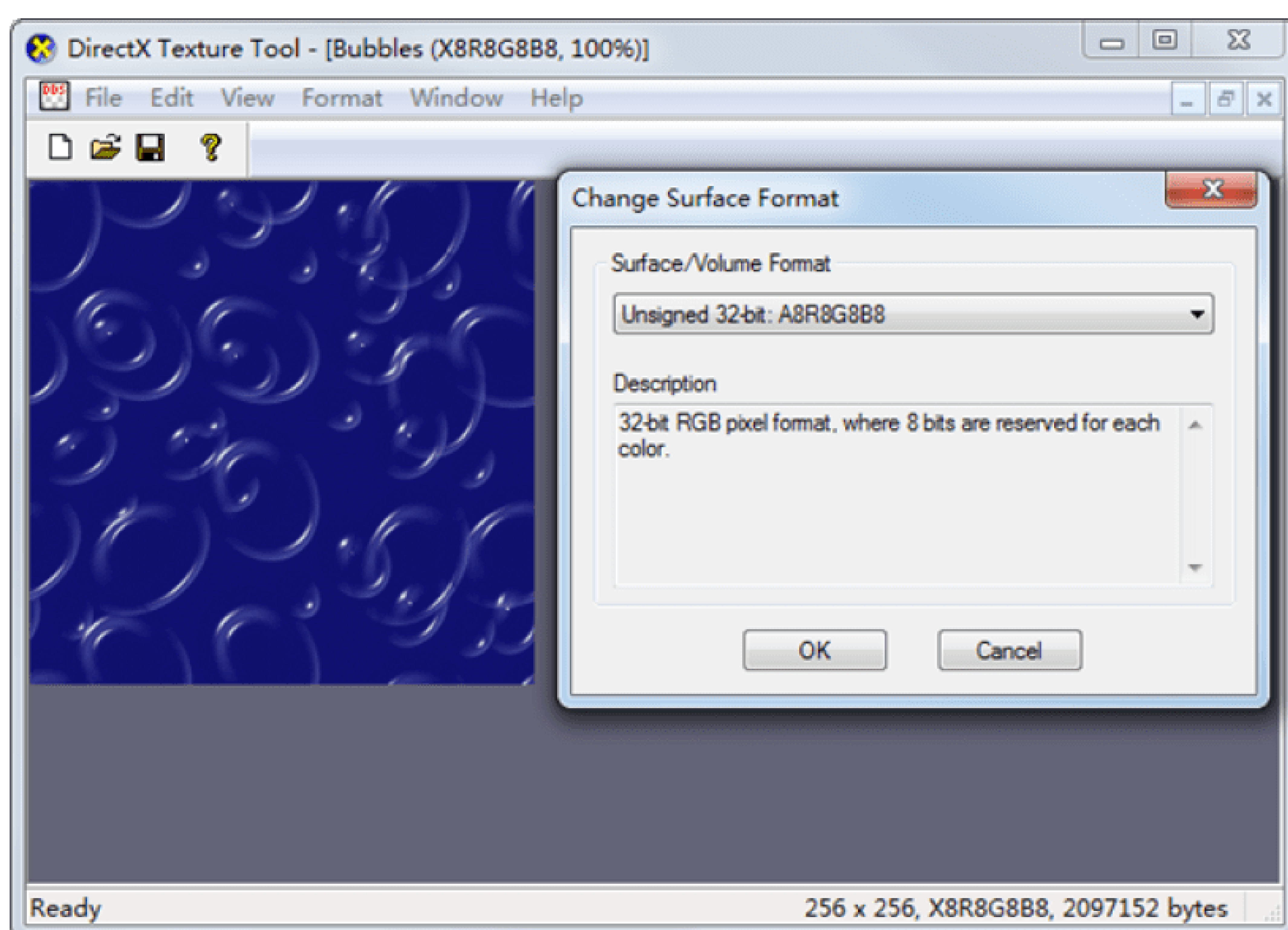


图 27-26　设置图像颜色格式

（4）选择 File|Open onto alpha channel of this texture 命令，选择要显示的图像，打开后效果如图 27-27 所示。

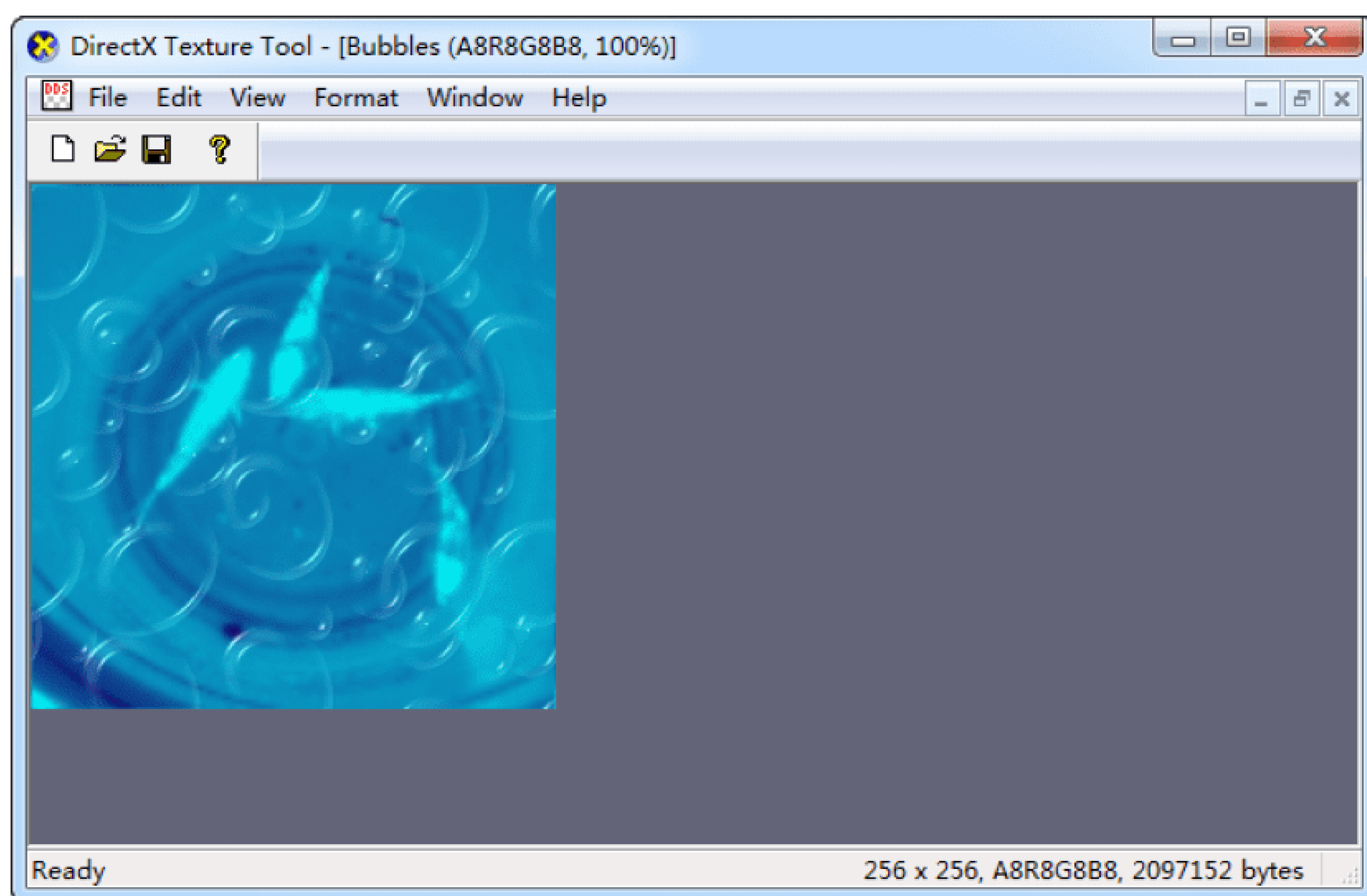


图 27-27　颜色混合效果

（5）选择 File|Save 命令，将叠加在一起的源图信息和 Alpha 通道信息保存为 DDS 格式的 Bubbles.dds。图 27-27 中显示的图像就是 Alpha 颜色混合的一个典型例子。

27.8.3　利用 ID3DXSprite 实现颜色透明

颜色透明是 Alpha 混合中典型的应用，Direct3D 提供了 ID3DXSprite 接口，可以实现

复杂的图像显示方式，称为精灵图像，可以实现颜色透明的效果。通过 Direct3D 设备对象的 D3DXCreateSprite()函数可以创建 ID3DXSprite 接口，其函数原型为：

```
HRESULT D3DXCreateSprite(
  LPDIRECT3DDEVICE9 pDevice,          //指向 Direct3D 设备对象的指针
  LPD3DXSPRITE * ppSprite);           //指向 ID3DXSprite 接口对象的指针
```

使用 ID3DXSprite 的过程如下。

（1）调用 ID3DXSprite 对象的 Begin()函数，启动设备准备绘制精灵图像，可以实现渲染、Alpha 混合和 sprite 变换以及排序。其函数原型为：

```
HRESULT Begin(DWORD Flags);           //指定渲染精灵图像的选项
```

使用此参数可以实现复杂的精灵图像，此处就不再详细说明了。

（2）每次显示精灵图像时，调用 ID3DXSprite 对象的 Draw()函数，此函数可以重复调用。其函数原型为：

```
HRESULT Draw(
  LPDIRECT3DTEXTURE9 pTexture,        //表示精灵图像的纹理对象
  CONST RECT * pSrcRect,              //表示用在此精灵图像上的源纹理图像的区域
  CONST D3DXVECTOR3 * pCenter,        //指定精灵图像的中心
  CONST D3DXVECTOR3 * pPosition,      //表示精灵的位置
  D3DCOLOR Color);                    //颜色和 Alpha 通道颜色使用此值混合
```

要缩放、旋转或变换精灵，则在调用 ID3DXSprite 对象的 Draw 之前，需要调用 SetTransform()函数进行相应的变换。

（3）要将精灵图像显示在设备上，需要调用 ID3DXSprite 对象的 End()函数和 Flush()函数。

其中，ID3DXSprite 对象的 Begin()函数和 End()函数是一对配套函数，并且必须在 IDirect3DDevice9 对象的 BeginScene()函数和 EndScene()函数之间调用。具体代码请参考 DirectXSDK，这里就不再赘述。

27.8.4　利用 Alpha 测试实现颜色透明

除了使用 ID3DXSprite 对象，还可以使用渲染管道流水线的 Alpha 测试功能来实现颜色透明，这种方法比使用 ID3DXSprite 对象要简单。渲染管道流水线的 Alpha 测试功能是在渲染图形像素颜色前，首先判断 Alpha 颜色值，如果 Alpha 颜色值满足指定的条件，则对应的像素颜色值会在屏幕表面绘制，否则会忽略掉对应的像素颜色值，不进行绘制。

（1）当需要产生透明效果的图像时，首先需要将要透明的部分镂空，此时，可以通过调用 D3DXCreateTextureFromFileEx()函数，创建背景为黑色的纹理对象，其 Alpha 颜色值为 0。当开始渲染管道流水线的 Alpha 测试时，指定 Alpha 值大于或等于某个值的像素点在屏幕上显示，否则不显示。这样就可以将需要做成透明效果的部分不显示。

（2）创建完透明黑色的纹理对象后，就可以将此纹理对象表示的图像透明地显示在其他图像上，之前要调用 SetRenderState()函数启动 Alpha 测试，并设置 Alpha 的参考值，代码如下：

```
m_d3dDevice ->SetRenderState(D3DRS_ALPHATESTENABLE, true);
```

```
m_d3dDevice ->SetRenderState(D3DRS_ALPHAREF, 0x01);
```

上面代码中的 D3DRS_ALPHATESTENABLE 值设置为 true，表示启动 Alpha 测试，D3DRS_ALPHAREF 值设置为 0x01，表示 Alpha 大于或等于 1 的像素都不显示，以达到透明的效果。

（3）接着设置 3DRS_ALPHAFUNC 状态为 D3DCMPFUNC 的枚举值，决定测试将使用比较方式。

```
typedef enum D3DCMPFUNC
{
    D3DCMP_NEVER = 1,                 //总是测试失败
    D3DCMP_LESS = 2,                  //接收 Alpha 值小于当前像素值的颜色
    D3DCMP_EQUAL = 3,                 //接收 Alpha 值等于当前像素值的颜色
    D3DCMP_LESSEQUAL = 4,             //接收 Alpha 值小于或等于当前像素值的颜色
    D3DCMP_GREATER = 5,               //接收 Alpha 值大于当前像素值的颜色
    D3DCMP_NOTEQUAL = 6,              //接收 Alpha 值不等于当前像素值的颜色
    D3DCMP_GREATEREQUAL = 7,          //接收 Alpha 值大于或等于当前像素值的颜色
    D3DCMP_ALWAYS = 8,                //总是测试成功
    D3DCMP_FORCE_DWORD = 0x7fffffff,
} D3DCMPFUNC, *LPD3DCMPFUNC;
```

如下代码是设置在 Alpha 测试时，使用的比较方式为接收 Alpha 值大于当前像素值的颜色。

```
m_d3dDevice ->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);
```

根据上面的步骤设置完毕后，就可以像渲染图像的方法一样，正常显示图像。具体代码请参考 DirectXSDK，这里就不再赘述。

27.9　XFile 网格的应用

XFile 是一种使用模板定义数据内容的文件格式，在显示三维图像时，会将物体表面分割成多个三角形，这种结构也称为网格，使用 Xfile 可以定义网格数据，从而快速地读写三维图像的数据。本节将介绍 XFile 网格的应用。

27.9.1　.x 文件的基本格式

在使用.x 文件之前，首先使用 Direct3D 提供的浏览.x 文件的工具来看看 XFile 文件的概念。选择“开始”|“程序”|Microsoft DirectX SDK（August 2008）|DirectX Utilities|DirectX Viewer 命令，打开 DirectX Viewer 工具。选择 File|Open 命令，打开 Open 对话框，从中选择要打开的.x 文件。此处打开 car.x 文件，如图 27-28 所示。

图 27-28 中，显示了一个绿色的小汽车，鼠标左键按下图像任意位置，然后拖动，会立体地旋转汽车。这就是 XFile 网格的一个典型应用，使用网格文件可以保存三维图像的数据，实现指定三维图像的纹理图案和顶点坐标等功能。为了理解.x 文件的格式，下面以一个立方体的.x 文件为例，讲解.x 文件的格式。下面是立方体.x 文件的内容。

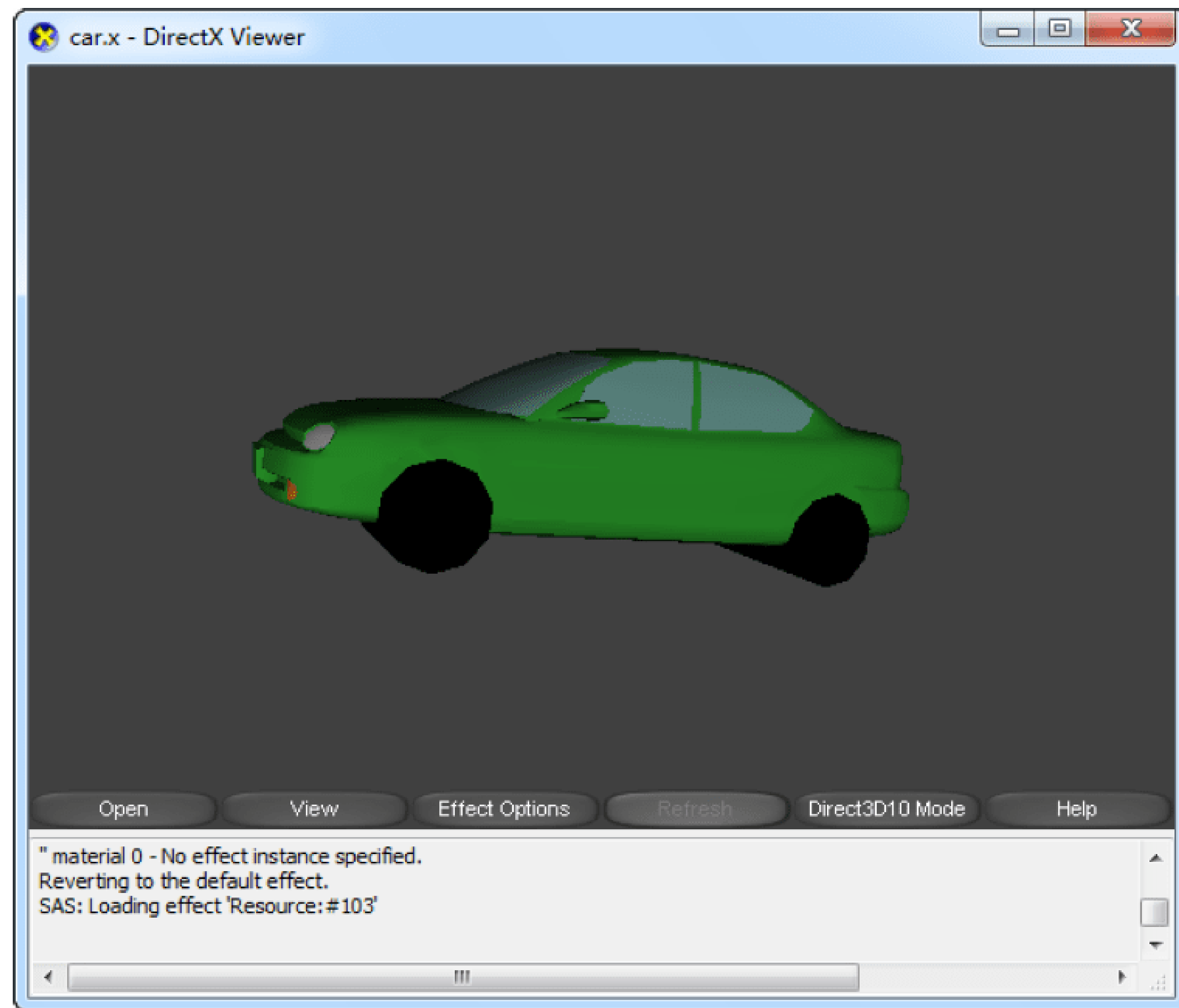


图 27-28　.x 文件示例

```
01  Material RedMaterial
02  {
03      1.000000;0.000000;0.000000;1.000000;;//R = 1.0, G = 0.0, B = 0.0
04      0.000000;
05      0.000000;0.000000;0.000000;;
06      0.000000;0.000000;0.000000;;
07  }
08  Material GreenMaterial
09  {
10      0.000000;1.000000;0.000000;1.000000;;//R = 0.0, G = 1.0, B = 0.0
11      0.000000;
12      0.000000;0.000000;0.000000;;
13      0.000000;0.000000;0.000000;;
14  }
15  Mesh CubeMesh
16  {
17      //定义具有 8 个顶点和 12 个三角形面的网格
18      //使用可选数据对象指定材质、法向量和纹理坐标
19      8;                                      //8 个顶点
20      1.000000;1.000000;-1.000000;,           //顶点 1
21          -1.000000;1.000000;-1.000000;,      //顶点 2
22          -1.000000;1.000000;1.000000;,       //顶点 3
23          1.000000;1.000000;1.000000;,        //顶点 4
24          1.000000;-1.000000;-1.000000;,      //顶点 5
25          -1.000000;-1.000000;-1.000000;,     //顶点 6
26          -1.000000;-1.000000;1.000000;,      //顶点 7
27          1.000000;-1.000000;1.000000;;       //顶点 8
28      12;                                     //12 个三角形面
29      3;0,1,2;,                               //第一个面具有 3 个顶点
30          3;0,2,3;,                           //第二个面具有 3 个顶点
31          3;0,4,5;,                           //第三个面具有 3 个顶点
32          3;0,5,1;,                           //第四个面具有 3 个顶点
```

```
33          3;1,5,6;,                               //第五个面具有 3 个顶点
34          3;1,6,2;,                               //第六个面具有 3 个顶点
35          3;2,6,7;,                               //第七个面具有 3 个顶点
36          3;2,7,3;,                               //第八个面具有 3 个顶点
37          3;3,7,4;,                               //第九个面具有 3 个顶点
38          3;3,4,0;,                               //第十个面具有 3 个顶点
39          3;4,7,6;,                               //第十一个面具有 3 个顶点
40          3;4,6,5;;                               //第十二个面具有 3 个顶点
41      //下面是可选部分，可以嵌套其他模板
42      MeshMaterialList
43      {
44              2;                              //使用的材质数目
45              12;                             //每个材质对每个三角形面
46              0,                              //第一个三角形使用第一个材质
47              0,                              //第二个三角形使用第一个材质
48              0,                              //第三个三角形使用第一个材质
49              0,                              //第四个三角形使用第一个材质
50              0,                              //第五个三角形使用第一个材质
51              0,                              //第六个三角形使用第一个材质
52              0,                              //第七个三角形使用第一个材质
53              0,                              //第八个三角形使用第一个材质
54              1,                              //第九个三角形使用第二个材质
55              1,                              //第十个三角形使用第二个材质
56              1,                              //第十一个三角形使用第二个材质
57              1;;                             //第十二个三角形使用第二个材质
58          {RedMaterial}                       //定义第一个材质为红色材质
59          {GreenMaterial}                     //定义第二个材质为绿色材质
60      }
61      MeshNormals
62      {                                               //定义法向量
63          8;                                          //分别定义 8 个顶点的法向量
64              0.333333;0.666667;-0.666667;,           //第一个顶点的法向量
65              -0.816497;0.408248;-0.408248;,          //第二个顶点的法向量
66              -0.333333;0.666667;0.666667;,           //第三个顶点的法向量
67              0.816497;0.408248;0.408248;,            //第四个顶点的法向量
68              0.666667;-0.666667;-0.333333;,          //第五个顶点的法向量
69              -0.408248;-0.408248;-0.816497;,         //第六个顶点的法向量
70              -0.666667;-0.666667;0.333333;,          //第七个顶点的法向量
71              0.408248;-0.408248;0.816497;;           //第八个顶点的法向量
72          12;                                         //分别定义 12 个面的法向量
73              3;0,1,2;,                               //第一个面的法向量
74              3;0,2,3;,                               //第二个面的法向量
75              3;0,4,5;,                               //第三个面的法向量
76              3;0,5,1;,                               //第四个面的法向量
77              3;1,5,6;,                               //第五个面的法向量
78              3;1,6,2;,                               //第六个面的法向量
79              3;2,6,7;,                               //第七个面的法向量
80              3;2,7,3;,                               //第八个面的法向量
81              3;3,7,4;,                               //第九个面的法向量
82              3;3,4,0;,                               //第十个面的法向量
83              3;4,7,6;,                               //第十一个面的法向量
84              3;4,6,5;;                               //第十二个面的法向量
85      }
86      MeshTextureCoords
```

```
87      {                    //定义网格纹理坐标
88          8;                                  //定义 8 个顶点的纹理坐标
89          0.000000;1.000000;                  //第一个顶点的纹理坐标
90          1.000000;1.000000;                  //第二个顶点的纹理坐标
91          0.000000;1.000000;                  //第三个顶点的纹理坐标
92          1.000000;1.000000;                  //第四个顶点的纹理坐标
93          0.000000;0.000000;                  //第五个顶点的纹理坐标
94          1.000000;0.000000;                  //第六个顶点的纹理坐标
95          0.000000;0.000000;                  //第七个顶点的纹理坐标
96          1.000000;0.000000;;                 //第八个顶点的纹理坐标
97      }
98  }
```

上面的例子，首先使用 Material 模板定义了两个材质，一种是红色材质，一种是绿色材质。然后使用 Mesh 定义了一个名称为 CubeMesh 的网格，指定了网格具有 8 个顶点和 12 个三角形面，并使用 MeshMaterialList 定义了网格使用的材质，使用 MeshNormals 定义了网格的法向量，包括顶点法向量和面法向量。最后，使用 MeshTextureCoords 定义了网格的纹理坐标。

从中可以看出，.x 文件是使用模板对数据进行定义的，并且模板之间是可以嵌套的，如上面的 Material 是材质模板，Mesh 是网格模板，MeshMaterialList 是网格使用的材质列表模板，MeshNormals 是网格的法向量模板，MeshTextureCoords 是定义网格纹理坐标的模板。模板的原型定义如下：

```
template <template-name> {      //模板定义
    <GUID>                      //全局标志符
    <member 1>                  //成员 1
    ...                         //此处代码省略
    < membern>                  //成员 n
   [restrictions]               //其他模板对象
}
```

其中，template-name 部分使用定义的模板名称来替换，GUID 是全局唯一标识符，使用相应的工具可以生成，member1～membern 是定义的模板中的成员，restrictions 中可以调用其他模板对象。为了简化工作，Direct3D 中预定义了部分模板，用户可以直接使用这些模板来定义数据。

27.9.2　.x 文件的数据装入

Direct3D 设备对象提供了操作.x 文件的 DirectXFile 接口，可以用来装载.x 文件、保存.x 文件以及向.x 文件中增加纹理和动画等操作。本小节介绍如何装入.x 文件的数据，其步骤如下。

（1）调用 DirectXFileCreate()函数创建 IDirectXFile 对象。

（2）如果要载入的模板是 DirectX 文件中存在的，则使用 IDirectXFile::RegisterTemplates()函数注册这些模板。

（3）使用 IDirectXFile::CreateEnumObject()函数创建一个 IDirectXFileEnumObject 枚举对象。

（4）循环处理文件中的对象，对于每个对象，执行如下操作。

- ❑ 使用 IDirectXFileEnumObject::GetNextDataObject()函数获取每个 IDirectXFileData 对象。
- ❑ 使用 IDirectXFileData::GetType()函数获取数据类型。
- ❑ 使用 IDirectXFileData::GetData()函数装载数据。
- ❑ 如果对象具有可选成员，使用 IDirectXFileData::GetNextObject()函数获取可选成员。
- ❑ 释放 IDirectXFileData 对象。

（5）释放 IDirectXFileEnumObject 对象。

（6）释放 IDirectXFile 对象。

除了这种装载.x 文件数据的方法外，Direct3D 还提供了 D3DXLoadMeshFromX()函数，可以直接从 DirectX 的.x 文件中装载网格。其函数原型为：

```
HRESULT D3DXLoadMeshFromX(
  LPCTSTR pFilename,                     //指定.x 文件名称
  DWORD Options,                         //指定创建网格的选项
  LPDIRECT3DDEVICE9 pD3DDevice,
  //指定与网格相连的设备对象 IDirect3DDevice9 的指针
  LPD3DXBUFFER * ppAdjacency,            //包含相连数据的缓冲区
  LPD3DXBUFFER * ppMaterials,            //指向包含材质数据的缓冲区
  LPD3DXBUFFER * ppEffectInstances,      //指向包含效果实例的缓冲区
  DWORD * pNumMaterials,//返回 ppMaterials 数组中 D3DXMATERIAL 结构的个数的指针
  LPD3DXMESH * ppMesh);                  //存放载入的网格
```

27.9.3　Mesh 数据的处理

加载完.x 文件后，Mesh 的数据存储在顶点缓冲区中，顶点信息存储在顶点索引缓冲区中和材质缓冲区中等。在这些数据缓冲区中，可以根据需要对 Mesh 数据进行处理和优化。其中网格中的顶点数据信息存储在顶点缓冲区中，网格图形的各个三角形面的顶点构成信息存储在顶点缓冲区中。同时在属性缓冲区中存储了每个三角形面的网格子集编号，具有相同编号的三角形，使用相同的材质和纹理渲染。

Mesh 数据最重要的处理就是进行 Mesh 网格渲染，其方法为使用循环结构，循环的次数为材质数目。对于每种材质，设置渲染管道线的材质和纹理对象，并将与 Mesh 网格中具有相同网格子集编号的三角形面渲染出来。这样，当所有的材质类型都渲染完后，整幅网格就渲染出来了。渲染指定网格子集编号使用 DrawSubset()函数，其函数原型为：

```
HRESULT DrawSubset( DWORD AttribId);         //指定要渲染的网格的子集的编号
```

此函数会将所有具有指定子集编号的网格三角形面进行渲染。

27.9.4　Mesh 数据的优化

装载入.x 文件的网格数据后，Direct3D 提供对网格子集数据进行优化的函数，可以提高渲染速度。如可以将顶点缓冲区中位于同一子集的顶点进行连续放置，减少三角形面的索引，提高渲染速度。Direct3D 的 ID3DXMesh 接口提供了 OptimizeInplace()函数对网格数

据进行各种优化，其函数原型为：

```
HRESULT OptimizeInplace(
  DWORD Flags,                  //指定优化类型
  CONST DWORD * pAdjacencyIn,//指定源网格的每个三角形面的 3 个相邻的面的顶点的数组
  DWORD * pAdjacencyOut,  //指定优化网格后，每个三角形面的 3 个相邻的面的顶点的数组
  DWORD * pFaceRemap,     //指定源网格面对应的优化网格中的三角形面
  LPD3DXBUFFER * ppVertexRemap);//包含每个新顶点对应的旧顶点
```

其中优化选项定义如下：

```
typedef enum D3DXMESHOPT{
  D3DXMESHOPT_COMPACT = 0x01000000,             //移除没有用的顶点和面
  D3DXMESHOPT_ATTRSORT = 0x02000000,            //提高 DrawSubset 性能
  D3DXMESHOPT_VERTEXCACHE = 0x04000000,         //增加顶点缓冲区单击率
  D3DXMESHOPT_STRIPREORDER = 0x08000000,        //最大化邻近三角形的长度
  D3DXMESHOPT_IGNOREVERTS = 0x10000000,         //仅优化面，不优化点
  D3DXMESHOPT_DONOTSPLIT = 0x20000000,
  //当属性排序时，不在属性组之间共享分离顶点
  D3DXMESHOPT_DEVICEINDEPENDENT = 0x40000000,  //影响顶点缓冲区大小
} D3DXMESHOPT, *LPD3DXMESHOPT;
```

使用此函数可以执行相应的优化选项。具体代码请参考 DirectXSDK，这里就不再赘述。

27.10 本 章 小 结

本章讲述了如何使用 DirectX 进行图形方面的开发。重点介绍了 DirectX SDK 的安装和配置、DirectX 图形开发的基本概念、基本三角形面和立体面的绘制，并介绍了图像材质、光照、纹理贴图和 Alpha 颜色的使用以及 XFile 网格的应用。本章的难点是如何将各种效果因素组合起来，绘制出逼真的图形效果。第 28 章将介绍如何使用 Visual Studio 2010 开发网络音频播放系统。

27.11 习 题

1．在 27.4.6 小节的示例中绘制了一个基本的三角形面，尝试对示例做如下修改：

（1）修改窗口的背景色为白色。

（2）修改显示矩形的大小。

（3）改变三角形 3 个角的颜色为：黄色、粉色、紫色。

【思路】理解 27.4.6 小节示例的执行流程，修改一些代码即可。

2．在 27.5.7 小节的示例中绘制了一个“金字塔”（即底部为四边形，其余 4 个面为三角形），尝试模仿该示例完成以下任务：

（1）绘制一个正方体，线条颜色为白色，窗体背景色为黑色。

（2）按下键盘上的 W、S、A、D 按键时可以控制正方体向上、向下、向左、向右旋转。

【思路】参考 27.5.7 小节的示例，修改绘制的立体图形，修改各个按键的作用。

第 7 篇　项目开发实战

第 28 章　网络音频播放系统

本章介绍如何使用 Visual Studio 2010 开发网络音频转换系统。此系统主要完成将网络上的音频资源转换成本地资源。本章主要涉及界面编程、网络编程、文件编程、多线程及线程同步编程、音频编程以及 COM 技术。在学习本章时，读者可以参考前面章节中的相关内容，深入理解这些编程知识。

28.1　系统分析与设计

本节对网络音频转换系统的分析和设计做了详细说明。28.1.1 小节分析了系统的功能，说明了系统实现的具体功能。28.1.2 小节介绍了设计的功能模块，并说明了每个模块实现的功能及实现方法。

28.1.1　功能描述

网络音频转换系统的功能就是完成将网络音频资源下载到本地并进行转换，同时可以实现同步播放。其实现的功能有：

- 下载网络音频资源到本地。
- 多线程同时下载。
- 下载的同时进行音频格式的转换。
- 实时播放网络音频资源。

28.1.2　功能模块设计

在本实例中，将有关窗体的界面类和音频处理类分开来，以实现功能模块的设计。主要有以下几部分。

- CBroadConvertDlg 类：主对话框类，实现程序的界面元素。
- CDlgItem 类：电台资源界面类，实现单个电台资源的界面元素。
- CAudioPlay 类：实现音频播放的类，包括启动音频播放、打开网络音频文件、停止音频播放、暂停音频播放和重启音频播放等功能。
- 其他相关功能：主要包括多线程安全的同步类、音频驱动和格式的相关函数以及用于处理网络音频下载的 CDRM 类。

下面将详细讲述这些类的实现。

28.2　界 面 实 现

本节介绍有关界面的实现。28.2.1 小节介绍界面的设计，包括其中的控件以及各个控件的功能。28.2.2 小节介绍界面初始化的实现。28.2.3 小节介绍具体实现的与界面有关的执行代码。

28.2.1　界面设计

网络音频转换系统的主界面是一个多页的标签对话框，会根据初始化信息来确定显示几页，这些初始化信息是要在系统中进行转换的电台的信息和个数。每有一个电台就会增加新页，所有的操作都是针对当前页面的电台信息的。如图 28-1 所示为每个电台资源下可以执行操作的界面。

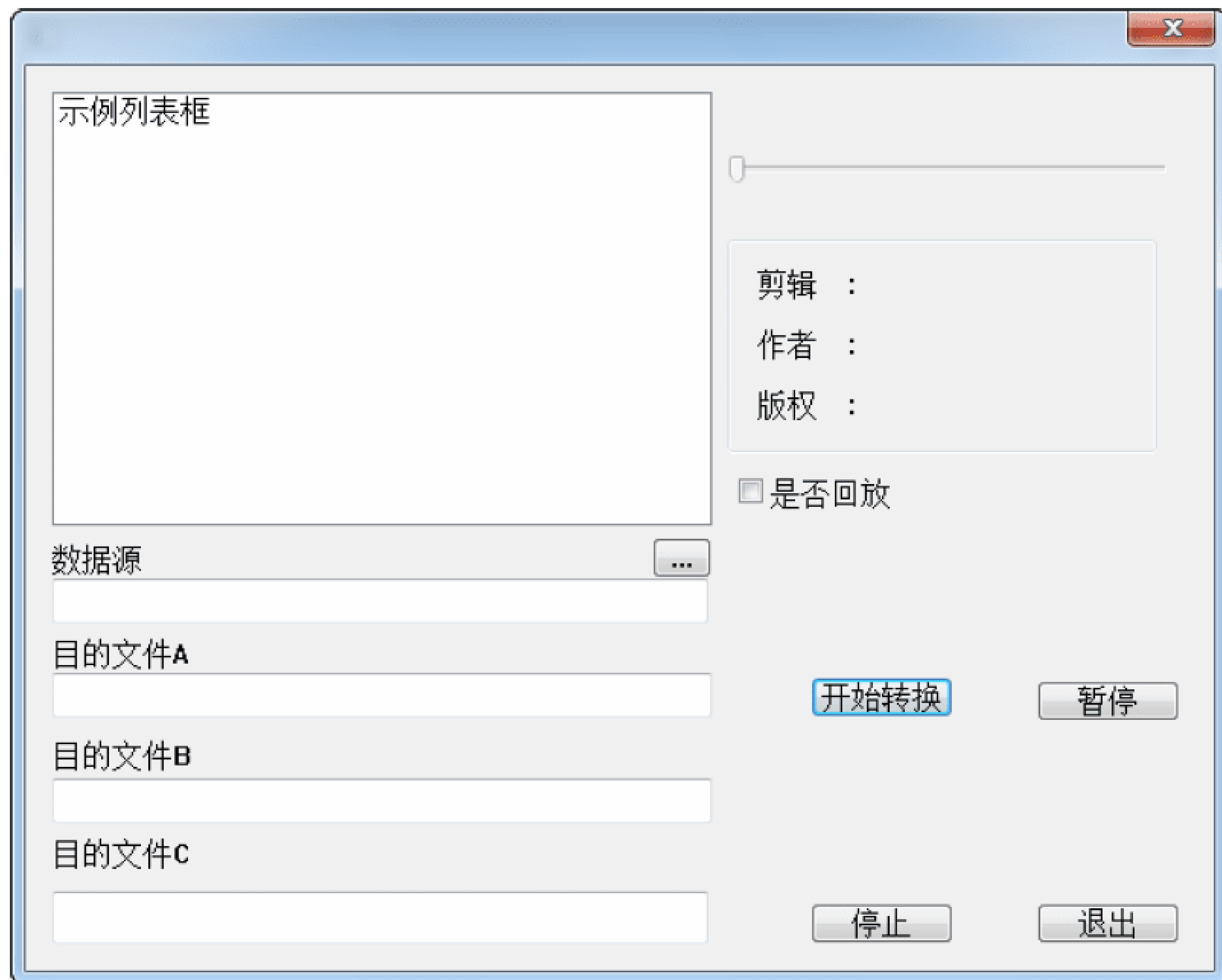


图 28-1　网络音频界面整体布局

图 28-1 所示左上角的列表框中列出了当前电台下的音频资源。“数据源”文本框中列出了电台音频的资源文件路径。“目的文件 A”文本框、“目的文件 B”文本框和“目的文件 C”文本框中分别显示了要将网络电台音频文件转换到本地的文件名，因为要支持网络音频的实时播放，而通过缓冲可以加速数据的显示，所以采取多文件写入的方式。界面上的滑块显示资源的播放进度。“剪辑”文本框中显示了资源的剪辑信息。“作者”文本框中显示了资源的作者。“版权”文本框中显示了当前资源的版权信息。“是否回放”复选框表示，在下载的同时是否同时回放。

单击“开始转换”按钮会启动从网络电台开始下载音频数据，并且如果选择了“是否

回放”复选框，则在下载的同时会进行数据的回放。“暂停”按钮可以暂停数据的下载和播放。“停止”按钮，则可以停止音频资源的下载和播放。单击“退出”按钮则会退出电台音频播放系统。

28.2.2　界面初始化

在进入网络音频播放程序后，需要进行初始化。在此程序中，主要是加载有效的电台资源和初始化播放的一些参数。代码如下：

```
01  void CBroadConvertDlg::OnButLoad()      //装载按钮执行的处理函数
02  {
03      CPropertySheet dlgPropertySheet("Simple Convert");
04      //创建存放电台资源信息的标签控件
05      CDlgItem stylePage[2];                //创建存放单个电台信息的窗体
06      //存放电台资源的网络地址
07      char cSource[2][MAX PATH]=
08      {
09          "mms://203.128.68.232/hit997",
10          "mms://218.244.243.30/fm91.5"
11      };
12      //存放电台名称
13      char cCaption[2][512]=
14      {
15          "电台 1--新城娱乐城",
16          "电台 6--fm91.5"
17      };
18      char cSaveFile[512];                      //定义默认的保存文件名
19      memset(cSaveFile, 0x00, sizeof(cSaveFile));
20      sprintf(cSaveFile, "%s",
21          "C:\\\\Documents and Settings\\\\www\\\\桌面\\\\EUCC\\\\
22          NEWVOICEFILE\\\\City\\\\Stations\\\\1002\\\\");
23      sprintf(cSaveFile, "%s", "F:\\\\NEWVOICEFILE\\\\City
24      \\\\Stations\\\\1002\\\\");
25      for (int i = 0 ; i < 2 ; i ++)
26      {
27          //分别为单个电台音频文件信息分配默认值
28          dlgPropertySheet.AddPage(&stylePage[i]);    //增加电台页
29          //设置选项
30          stylePage[i].m psp.dwFlags =
31              stylePage[i].m psp.dwFlags |PSP USETITLE;
32          stylePage[i].m_psp.pszTitle = (char*)&cCaption[i][0];
33          //设置电台页名称
34          char cSaveFileA[512] = {0};                 //定义文件 1
35          char cSaveFileB[512] = {0};                 //定义文件 2
36          char cSaveFileC[512] = {0};                 //定义文件 3
37          //保存 pcm 文件 1
38          sprintf(cSaveFileA, "%s%2.2dVoiceA.pcm", cSaveFile, i+1);
39          //保存 pcm 文件 2
40          sprintf(cSaveFileB, "%s%2.2dVoiceB.pcm", cSaveFile, i+1);
41          //保存 pcm 文件 3
```

```
42          sprintf(cSaveFileC, "%s%2.2dVoiceC.pcm", cSaveFile, i+1);
43          stylePage[i].InitFileName((char*)&cSource[i][0],
44              cSaveFileA, cSaveFileB,  cSaveFileC, false,
45              (char*)&cCaption[i][0]);
46      }
47      //设置存放电台信息的标签页面的样式并显示
48      dlgPropertySheet.m psh.dwFlags |= PSH NOAPPLYNOW ;
49      dlgPropertySheet.m_psh.pszCaption = "电台转换";  //页面标题
50      dlgPropertySheet.m_psh.nStartPage = 0;          //开始页面
51      dlgPropertySheet.DoModal();                     //显示页面
52  }
```

上面代码主要是初始化两个有效的电台资源，在各个标签页上显示。在页面上初始化存储文件的文件名。最后设置标签的样式，包括名称和起始页以及显示的样式。

28.2.3　界面代码

下面显示了初始化电台播放界面时的源代码：

```
01  BOOL CDlgItem::OnInitDialog()                        //Tab 页面初始化函数
02  {
03      CPropertyPage::OnInitDialog();                   //调用基类的函数
04      //Step1: 定义变量
05      HRESULT      hr       = S OK;
06      TCHAR        tszFileName[ MAX PATH ];
07      //Step2: 准备打开播放文件，创建并初始化音频播放器
08      SetCurrentStatus( READY );                       //设置当前状态
09      hHeap = HeapCreate(0, HEAPSIZE, 0);              //创建 heap
10      if (hHeap == NULL)
11          return false;                                //判断创建结果
12      g_pAudioplay = new CAudioPlay;                   //创建 CAudioPlay 对象
13      if( NULL == g pAudioplay )
14          return false;                                //判断创建结果
15      hr = g_pAudioplay->Init();                       //初始化音频播放对象
16      if( FAILED(hr) )
17          return false;                                //判断初始化结果
18      g_pAudioplay->pParDlg = this;                    //赋值对话框
19      m_editSource.SetWindowText( g_ptszFileName );
20      //设置音频源文本框中的内容
21      m editDesFileA.SetWindowText( g pSaveFileNameA );
22      //分别设置保存的文件名
23      m editDesFileB.SetWindowText( g pSaveFileNameB );
24      m_editDesFileC.SetWindowText( g_pSaveFileNameC );
25      m_CheckPlay.SetCheck( m_bPlayBack );     //设置是否选择播放复选框状态
26      m_ButPlay.SetFocus();                            //将焦点赋值给播放按钮
27      GetDlgItemText( IDC EDIT SOURCE, tszFileName, MAX PATH );
28      //Step3: 如果 filename 不为空，则使得 Play 按钮可用
29      if( _tcslen( tszFileName) > 0 )//判断获取的音频源文件长度是否大于 0
30      {
31          //如果是
32          m_ButPlay.EnableWindow( true );              //使得播放按钮可用
33          SetCurrentStatus( CLOSED );                  //设置当前状态为关闭
34      }
```

```
35      else
36         m ButPlay.EnableWindow( false );         //否则使得播放按钮不可用
37      SetCurrentStatus( READY );                  //设置当前状态为准备好
38      OnPlay();                                   //播放
39      SetTimer(200, 60000, NULL);                 //启动定时器
40      return true;                                //函数返回
41  }
```

在上面代码中，第一步定义了用到的变量。第二步准备打开播放文件，创建并初始化音频播放器，其中调用了 g_pAudioplay 对象的 Init()函数来初始化 CAudioPlay 类对象，此对象会在后面介绍。第三步，完成了界面控件的初始化，并启动工作定时器。下面代码显示了鼠标拖放滑条时的处理：

```
01  void CDlgItem::SetTimer(QWORD cnsTimeElapsed, QWORD cnsFileDuration)
02  {
03      if( g IsSeeking )
04         return;                                  //判断是否到达结尾
05      DWORD dwSeconds = 0;                        //定义存放秒的变量
06      TCHAR tszTime[20];                          //定义时间字符串
07      TCHAR tszTemp[10];                          //定义临时变量
08      UINT  nHours = 0;                           //定义小时
09      UINT  nMins = 0;                            //定义分钟
10      ZeroMemory( (void *)tszTime, sizeof( tszTime ) );//初始化时间变量
11      dwSeconds = ( DWORD )( cnsTimeElapsed / 10000000 );//计算秒值
12      nHours = dwSeconds / 60 / 60;               //计算小时
13      dwSeconds %= 3600;                          //计算秒
14      nMins = dwSeconds / 60;                     //计算分钟
15      dwSeconds %= 60;                            //计算秒
16      //计算鼠标释放位置所代表的时间值
17      if( 0 != nHours )
18      {
19         _stprintf( tszTemp, _T( "%d:" ), nHours );
20         _tcscat( tszTime, tszTemp );
21      }
22      _stprintf( tszTemp, _T( "%02d:%02d / " ), nMins, dwSeconds );
23      _tcscat( tszTime, tszTemp );
24      nHours = 0;
25      nMins = 0;
26      dwSeconds = ( DWORD )( cnsFileDuration / 10000000 );
27      nHours = dwSeconds / 60 / 60;
28      dwSeconds %= 3600;
29      nMins = dwSeconds / 60;
30      dwSeconds %= 60;
31      if( 0 != nHours )
32      {
33          stprintf( tszTemp,  T( "%d:" ), nHours );
34          tcscat( tszTime, tszTemp );
35      }
36       stprintf( tszTemp,  T( "%02d:%02d" ), nMins, dwSeconds );
37      //格式化分和秒
38      _tcscat( tszTime, tszTemp );                //连接时间值
39      //将滑动条移动到鼠标拖放的位置，并发送命令，使音频播放定位到指定时间上
40      SendDlgItemMessage( IDC SLIDER , TBM SETPOS, true,
41         ( LONG )( cnsTimeElapsed / 10000 ) );
42      SendDlgItemMessage( IDC_DURATION, WM_SETTEXT, 0,
43         ( WPARAM )tszTime );
```

```
44      return;
45  }
```

上面代码中，第一步计算鼠标释放时的位置所代表的时间值，第二步将滑条移动过去，并发送消息通知音频播放到指定时间上。下面显示了当单击“开始转换”按钮时执行的代码：

```
01  void CDlgItem::OnOpen()                          //“开始转换”按钮的执行函数
02  {
03      TCHAR tszFileName[ MAX_PATH ];               //定义文件名变量
04      GetDlgItemText( IDC_EDIT_SOURCE, tszFileName, MAX_PATH );
05      //获取文件名
06      if( _tcslen( tszFileName) > 0 )              //如果文件名长度大于 0
07      {
08          m_ButPlay.EnableWindow( true );          //启动播放按钮
09          SetCurrentStatus( CLOSED );              //设置当前状态为关闭
10      }
11      else
12          m_ButPlay.EnableWindow( false );         //否则播放按钮不可用
13      SetCurrentStatus( READY );                   //设置当前状态为准备好
14      return ;
15  }
```

在上面代码中，如果转换文件存在，则“开始转换”按钮可用，否则不可用。然后设置当前的状态为准备好（READY），定时器处理函数就会执行相应的操作。下面显示了单击“暂停”按钮时的执行代码：

```
01  void CDlgItem::OnPause()                     //“暂停”按钮的执行函数
02  {
03      HRESULT hr = S_OK;                       //定义结果句柄
04      if( NULL != g_pAudioplay )               //判断音频播放对象是否有效
05      {
06          hr = g_pAudioplay->Pause();          //设置音频播放对象为暂停
07          if( FAILED( hr ) )        //判断设置结果，如果失败，则输出错误信息
08          {
09              TCHAR tszErrMesg[128];
10              _stprintf(tszErrMesg,_T("Unable to Pause (hr=%#X)"),hr );
11              InsertLog( tszErrMesg );
12          }
13          else
14              SetCurrentStatus( PAUSE );       //设置当前状态为暂停
15      }
16  }
```

上面代码先调用 g_pAudioplay 对象的 Pause()函数暂停音频播放器的播放操作，如果失败，则会设置工作状态为暂停（PAUSE）。下面显示了单击“播放”按钮时执行的代码：

```
01  void CDlgItem::OnPlay()                       //“播放”按钮执行函数
02  {
03      HRESULT hr = S_OK;                        //定义结果句柄
04      if( NULL == g_pAudioplay )
05          return;                               //判断音频播放对象是否有效
06      m_bPlayBack = m_CheckPlay.GetCheck();     //获取播放复选框的取值
07      switch( g_Status )                        //检查播放器先前的状态
08      {
```

```
09      case PAUSE:                              //如果先前是暂停
10          hr = g_pAudioplay->Resume();         //恢复播放
11          if( FAILED( hr ) )              //判断处理结果，如果失败，则输出错误信息
12          {
13              TCHAR tszErrMesg[ 128 ];
14               stprintf( tszErrMesg,  T( "Unable to resume (hr=%#X)" ), hr );
15              InsertLog(tszErrMesg);
16          }
17          else
18              SetCurrentStatus( PLAY );        //设置当前状态为播放
19          break;
20      case STOP:                               //如果先前是停止
21          break;
22      case CLOSED:                             //如果先前是关闭
23          SetCurrentStatus( OPENING );         //设置当前状态为打开
24          GetDlgItemText(IDC EDIT SOURCE, g ptszFileName, MAX PATH );
25          //获取文件名
26          TCHAR *ptszTemp = g_ptszFileName;    //处理文件名取值
27          while( *ptszTemp == _T(' ') ) ptszTemp++;
28          if( g ptszFileName != ptszTemp )     //如果有效，则发送文件名消息
29          {
30              memmove( g_ptszFileName, ptszTemp,
31                  sizeof( TCHAR ) * ( _tcslen( ptszTemp ) + 1 ) );
32              SendDlgItemMessage( IDC EDIT SOURCE, WM SETTEXT,
33                  0, ( WPARAM )g ptszFileName );
34          }
35          ptszTemp = g_pSaveFileNameA;         //获取保存文件名 A
36          while( *ptszTemp == _T(' ') )
37          {
38              ptszTemp++;
39          }
40          if( g_pSaveFileNameA != ptszTemp )   //处理保存文件名 A，并发送消息
41          {
42              memmove( g pSaveFileNameA, ptszTemp,
43                  sizeof( TCHAR ) * ( _tcslen( ptszTemp ) + 1 ) );
44              SendDlgItemMessage( IDC EDIT DESFILEA, WM SETTEXT,
45                  0, ( WPARAM )g pSaveFileNameA );
46          }
47          GetDlgItemText(IDC EDIT DESFILEB,
48              g pSaveFileNameB, MAX PATH);
49          ptszTemp = g_pSaveFileNameB;         //获取保存文件名 B
50          while( *ptszTemp == _T(' ') )
51          {
52              ptszTemp++;
53          }
54          if( g_pSaveFileNameB != ptszTemp )   //处理保存文件名 B，并发送消息
55          {
56              memmove( g pSaveFileNameB, ptszTemp,
57                  sizeof( TCHAR ) * ( _tcslen( ptszTemp ) + 1 ) );
58              SendDlgItemMessage( IDC EDIT DESFILEB, WM SETTEXT,
59                  0, ( WPARAM )g pSaveFileNameB );
60          }
61          GetDlgItemText(IDC EDIT DESFILEC,g pSaveFileNameC,MAX PATH);
62          ptszTemp = g_pSaveFileNameC;         //获取保存文件名 C
63          while( *ptszTemp == _T(' ') )
64              ptszTemp++;
```

```
65          if( g pSaveFileNameC != ptszTemp )          //获取保存文件名 C
66          {
67              memmove( g pSaveFileNameC, ptszTemp,
68                  sizeof( TCHAR ) * ( _tcslen( ptszTemp ) + 1 ) );
69              SendDlgItemMessage( IDC_EDIT_DESFILEC, WM_SETTEXT,
70                  0, ( WPARAM )g pSaveFileNameC );
71          }
72  #ifndef UNICODE                                         //文件名 Unicode 码处理
73          {
74              WCHAR pwszFileName[ MAX PATH ];             //定义多字符集字符串变量
75              //将文件名转换为多字节数据集
76              if( 0 == MultiByteToWideChar( CP_ACP, 0,
77                  g ptszFileName, -1,pwszFileName, MAX PATH ) )
78              {
79                  SetCurrentStatus( CLOSED );
80                  SetCurrentStatus( READY );
81                  break;
82              }
83              WCHAR pwszSaveFileNameA[ MAX_PATH ]; //定义多字符集字符串变量
84              //将文件名转换为多字节数据集
85              if( 0 == MultiByteToWideChar( CP ACP, 0,
86                  g pSaveFileNameA, -1,pwszSaveFileNameA, MAX PATH ) )
87              {
88                  SetCurrentStatus( CLOSED );
89                  SetCurrentStatus( READY );
90                  break;
91              }
92              WCHAR pwszSaveFileNameB[ MAX_PATH ]; //定义多字符集字符串变量
93              //将文件名转换为多字节数据集
94              if( 0 == MultiByteToWideChar( CP ACP, 0,
95                  g_pSaveFileNameB, -1,pwszSaveFileNameB, MAX_PATH ) )
96              {
97                  SetCurrentStatus( CLOSED );
98                  SetCurrentStatus( READY );
99                  break;
100             }
101             WCHAR pwszSaveFileNameC[ MAX_PATH ]; //定义多字符集字符串变量
102             if( 0 == MultiByteToWideChar( CP ACP, 0,
103                 g pSaveFileNameC, -1,pwszSaveFileNameC, MAX PATH ) )
104                 //将文件名转换为多字节数据集
105             {
106                 SetCurrentStatus( CLOSED );
107                 SetCurrentStatus( READY );
108                 break;
109             }
110             hr = g_pAudioplay->Open( pwszFileName,
111                 (const unsigned short*)&g_pSaveFileNameA,
112                 (const unsigned short*)
113                 &g pSaveFileNameB,
114                 (const unsigned short*)&g_pSaveFileNameC,
115                 m bPlayBack );
116             //音频播放对象打开文件
117         }
118 #else
119         //音频播放对象打开文件
120         hr = g pAudioplay->Open( g ptszFileName, g pSaveFileNameA,
121             g_pSaveFileNameB, g_pSaveFileNameC, m_bPlayBack );
```

```
122 #endif //UNICODE
123         if( FAILED( hr ) )                              //判断操作结果
124         {
125             SetCurrentStatus( CLOSED );                 //设置当前状态为关闭
126             SetCurrentStatus( READY );                  //设置当前状态为准备好
127         }
128         else
129         {
130             hr = g_pAudioplay->Start();                 //开始音频播放
131             if( FAILED( hr ) )          //判断设置结果，如果失败，则输出错误信息
132             {
133                 TCHAR tszErrMesg[ 128 ];
134                 _stprintf(tszErrMesg,_T("Unable to start (hr=%#X)"),hr);
135                 InsertLog( tszErrMesg );
136             }
137             else                                        //发送播放进度通知消息
138             {
139                 SendDlgItemMessage( IDC_SLIDER, TBM_SETRANGEMAX, true,
140                     (DWORD )( g_pAudioplay->GetFileDuration()/10000 ));
141                 LPTSTR ptszFile = tcsrchr(g_ptszFileName, T('\\') );
142                 //显示播放文件名
143                 if( NULL != ptszFile )
144                     SetWindowText( ptszFile + 1 );
145                 else
146                     SetWindowText( g_ptszFileName );
147             }
148         }
149         break;
150     }
151 }
```

上面代码处理了播放函数，根据原来的播放状态来处理现在的播放状态。调用了 g_pAudioplay 对象的 Start()函数来启动音频播放，Open()函数用来打开指定的文件进行播放，Resume()函数恢复音频播放。下面显示了单击“停止”按钮时的处理代码：

```
01  void CDlgItem::OnStop()                       //“停止”按钮的处理函数
02  {
03      HRESULT  hr = S_OK;                        //定义返回的结果变量
04      if( NULL != g_pAudioplay )                 //如果播放对象不为 NULL
05      {
06          SetCurrentStatus( STOPPING );          //设置播放状态为正在播放
07          hr = g_pAudioplay->Stop();             //停止音频播放
08          if( FAILED( hr ) )          //判断设置结果，如果失败，则输出错误信息
09          {
10              SetCurrentStatus( g_Status );//设置音频播放状态
11              TCHAR tszErrMesg[128];             //输出提示信息
12               stprintf( tszErrMesg, T("Unable to Stop (hr=%#X)"),hr);
13              InsertLog( tszErrMesg );
14          }
15      }
16      return ;
17  }
```

上面代码调用了 g_pAudioplay 对象的 Stop()函数来停止音频的播放。下面显示了退出界面时的处理代码：

```
01  void CDlgItem::OnCancel()                    //取消按钮的处理函数
02  {
03      if( NULL != g_pAudioplay )               //判断音频播放对象是否有效
04      {
05          g_pAudioplay->Exit();                //退出音频播放对象
06          g_pAudioplay->Release();             //释放音频播放对象
07      }
08      if (hHeap != NULL)
09          HeapDestroy(hHeap);                  //释放 heap
10      CPropertyPage::OnCancel();               //调用基类的取消处理函数
11  }
```

上面代码在退出界面时，调用 g_pAudioplay 对象的 Exit()函数退出，并调用 Release()函数释放定义的 hHeap 对象。下面为定时器的执行代码，定时器的功能就是定期执行播放函数。

```
01  void CDlgItem::OnTimer(UINT nIDEvent)        //定时器处理函数
02  {
03      if (nIDEvent == 200)                     //定时播放
04      {
05          OnPlay();
06      }
07      CPropertyPage::OnTimer(nIDEvent);
08  }
```

28.3　核 心 实 现

本章介绍了完成程序的核心实现。28.3.1 小节介绍有关线程同步类的实现。28.3.2 小节介绍音频驱动函数，如音频格式枚举、音频驱动枚举以及查找支持指定音频格式的音频驱动等。28.3.3 小节介绍音频播放类 CAudioPlay 的声明。

28.3.1　线程同步类

因为在此程序中使用了多线程编程技术，所以需要处理线程的同步。本例中，通过自定义同步类来完成多线程之间的同步，避免发生资源冲突。如下代码为同步类的声明：

```
01  class CSync
02  {
03      HANDLE      m_sync;                  //同步句柄
04  public:
05      CSync ();                            //构造函数
06      ~CSync ();                           //析构函数
07      CSync (CSync& s);                    //带初始变量的构造函数
08      CSync& operator= (CSync& s);         //赋值重载符
09      void Enter () const;                 //进入同步对象
10      void Leave () const;                 //退出同步对象
11  };
```

上面代码定义了同步类，其中定义了 HANDLE 类型的同步对象 m_sync，并声明了同步类的构造函数、析构函数以及进入关键段和退出关键段的函数。下面显示了这些函数的

实现代码：

```
01  CSync::CSync()                    //同步类的构造函数
02  {
03      m sync = CreateMutex (NULL, false, NULL);
04      if (m sync == NULL)
05          throw CError (1001);
06  }
07  CSync::~CSync()                   //同步类的析构函数
08  {
09      if (m_sync != NULL)           //关闭句柄，并清空同步对象
10      {
11          CloseHandle (m sync);
12          m sync = NULL;
13      }
14  }
15  void CSync::Enter () const        //进入关键段
16  {
17      WaitForSingleObject (m_sync, INFINITE);
18  }
19  void CSync::Leave () const        //释放关键段
20  {
21      ReleaseMutex (m sync);
22  }
```

上面代码进入关键段函数 Enter()，调用 WaitForSingleObject()函数等待关键段对象并传入参数 INFINITE，表示不使用超时时间。退出关键段函数 Leave()，调用 ReleaseMutex 释放关键段资源。

28.3.2　音频驱动函数

本小节使用一组音频驱动函数来枚举和处理音频驱动。下面列出了音频格式枚举的回调函数代码。

```
01  BOOL CALLBACK find format enum(HACMDRIVERID hadid,
02      LPACMFORMATDETAILS pafd, DWORD dwInstance, DWORD fdwSupport)
03      //枚举音频格式
04  {
05      FIND_DRIVER_INFO* pdi;                      //查找驱动信息
06      pdi = (FIND_DRIVER_INFO*) dwInstance;       //赋值
07      if (pafd->dwFormatTag == (DWORD)pdi->wFormatTag)
08      //如果查找到指定格式，则退出枚举
09      {
10          Pdi->hadid = hadid;                     //赋值 hadid
11          return false;                           //停止枚举
12      }
13      return true;                                //继续枚举
14  }
```

此函数通过枚举传入的格式与应用实例的格式值进行比较，如果两者相符，则返回 false，表示不再继续枚举；否则，返回 true，继续枚举。如此循环，直到枚举结束或查找到符合条件的音频格式。下面列出了音频驱动枚举的回调函数代码：

```
01  BOOL CALLBACK find_driver_enum(HACMDRIVERID hadid,
```

```
02      DWORD dwInstance, DWORD fdwSupport)     //驱动枚举回调函数
03  {
04      //变量定义
05      ACMFORMATDETAILS    fd;                 //格式详细信息变量
06      FIND_DRIVER_INFO*       pdi = NULL;     //驱动信息变量
07      WAVEFORMATEX*           pwf = NULL;     //WAVE 格式变量
08      HACMDRIVER              had = NULL;     //ACM 驱动变量
09      MMRESULT            mmr;                //操作结果变量
10      DWORD                   dwSize = 0;     //长度变量
11      BOOL                bSuccess = true;    //是否成功
12      pdi  = (FIND_DRIVER_INFO*) dwInstance;  //赋值驱动信息
13      mmr = acmDriverOpen(&had, hadid, 0);    //打开 ACM 驱动
14      if ( mmr != MMSYSERR_NOERROR )          //判断操作结果
15      {
16          bSuccess = false;
17          goto HappenError;
18      }
19      //获取 acm 信息
20      mmr = acmMetrics( (HACMOBJ)had,
21          ACM_METRIC_MAX_SIZE_FORMAT, &dwSize );
22      if ( dwSize < sizeof( WAVEFORMATEX ) )      //判断获取的信息长度
23          dwSize = sizeof( WAVEFORMATEX );
24      pwf = (WAVEFORMATEX*) malloc(dwSize);       //分配空间
25      memset(pwf, 0, dwSize);                     //初始化变量
26      pwf->cbSize = LOWORD(dwSize) - sizeof(WAVEFORMATEX);//赋值长度
27      pwf->wFormatTag = pdi->wFormatTag;          //赋值格式
28      memset(&fd, 0, sizeof(fd));                 //初始化变量
29      fd.cbStruct = sizeof(fd);                   //赋值结构长度
30      fd.pwfx = pwf;                              //赋值 pwfx
31      fd.cbwfx = dwSize;                          //赋值长度
32      fd.dwFormatTag = pdi->wFormatTag;           //赋值格式
33      mmr = acmFormatEnum(had, &fd,
34          find_format_enum, (DWORD)(VOID*)pdi, 0);
35      //枚举格式
36      if ( mmr != MMSYSERR NOERROR )              //判断操作结果
37      {
38          bSuccess = false;                       //赋值结果变量为 false
39          goto HappenError;
40      }
41      if ( pdi->hadid != NULL )                   //判断 hadid 是否有效
42      {
43          bSuccess = false;                       //赋值结果变量为 FALS
44          goto HappenError;
45      }
46  HappenError:                                    //枚举时发生错误
47      acmDriverClose(had, 0);                     //关闭 acm 驱动实例
48      SAFE_ARRAYDELETE( pwf );                    //释放 pwf
49      if ( bSuccess )
50          return true;                            //返回相应的操作结果
51      else
52          return false;
53  }
```

此函数打开驱动，并且枚举该驱动支持的音频格式。下面代码显示了查找支持指定格式的驱动：

```
01  HACMDRIVERID find driver(WORD wFormatTag)           //查找驱动
02  {
03      FIND_DRIVER_INFOfdi;                            //查找驱动信息
04      MMRESULT        mmr;                            //操作结果
05      fdi.hadid = NULL;                               //赋值 hadid
06      fdi.wFormatTag = wFormatTag;                    //赋值格式 Tag
07      mmr = acmDriverEnum( find driver enum,
08          (DWORD)(VOID*)&fdi, 0 );
09      //枚举 acm 驱动
10      if ( mmr != MMSYSERR_NOERROR )
11          return NULL; //判断操作结果
12      return fdi.hadid;                               //返回 hadid
13  }
```

下面代码显示查找到的第一个支持指定格式的驱动的信息：

```
01  WAVEFORMATEX* get_driver_format(HACMDRIVERID hadid,
02      WORD wFormatTag)
03  {                                               //获取驱动格式
04      HACMDRIVER  had = NULL;                     //定义 ACM 驱动句柄
05      MMRESULTmmr;                                //定义操作结果
06      DWORD       dwSize = 0;                     //定义长度
07      WAVEFORMATEX*       pwf = NULL;             //定义 pwf 变量
08      ACMFORMATDETAILSfd;                         //定义 ACM 格式详细信息变量
09      BOOL     bSuccess = true;                   //定义是否成功结果
10      mmr = acmDriverOpen(&had, hadid, 0);        //打开 acm 驱动
11      if ( mmr != MMSYSERR_NOERROR )              //判断操作结果
12      {
13          bSuccess = false;                       //赋值结果变量为 false
14          goto HappenError;
15      }
16      //获取 acm 的相关参数
17      mmr = acmMetrics((HACMOBJ)had,
18          ACM_METRIC_MAX_SIZE_FORMAT, &dwSize);
19      if ( mmr != MMSYSERR_NOERROR )              //判断操作结果
20      {
21          bSuccess = false;                       //赋值结果变量为 false
22          goto HappenError;
23      }
24      if (dwSize < sizeof(WAVEFORMATEX))          //判断 dwSize 值是否有效范围
25      {
26          dwSize = sizeof(WAVEFORMATEX);
27      }
28      pwf = (WAVEFORMATEX*) malloc(dwSize);       //申请存储空间
29      memset(pwf, 0, dwSize);                     //初始化 pwf 值
30      pwf->cbSize = LOWORD(dwSize) - sizeof(WAVEFORMATEX);
31      //赋值 cbSize 分量
32      pwf->wFormatTag = wFormatTag;               //赋值格式标记
33      memset(&fd, 0, sizeof(fd));                 //初始化变量
34      fd.cbStruct = sizeof(fd);                   //赋值 cbStruct 值
35      fd.pwfx = pwf;                              //赋值 pwf
36      fd.cbwfx = dwSize;                          //赋值 pwf 的长度
37      fd.dwFormatTag = wFormatTag;                //赋值格式标记
38      FIND_DRIVER_INFO fdi;                       //定义驱动查找信息变量
39      fdi.hadid = NULL;                           //初始化 hadid
40      fdi.wFormatTag = wFormatTag;                //赋值 format 标记
```

```
41      //acm 格式化枚举
42      mmr = acmFormatEnum( had, &fd,
43          find_format_enum, (DWORD)(VOID*)&fdi,0 );
44      if ( mmr != MMSYSERR_NOERROR )              //判断操作结果
45      {
46          bSuccess = false;                       //赋值操作结果为 false
47          goto HappenError;
48      }
49      if ( NULL == fdi.hadid )                    //判断操作结果
50      {
51          bSuccess = false;                       //赋值操作结果为 false
52          goto HappenError;
53      }
54  HappenError:                                    //当函数发生错误
55      acmDriverClose(had, 0);                     //关闭驱动查找
56      if ( bSuccess )                             //如果成功，则返回 pwf
57      {
58          return pwf;
59      }
60      else                                        //如果失败
61      {
62          SAFE_ARRAYDELETE( pwf );                //释放 pwf 变量
63          return NULL;                            //返回 NULL
64      }
65  }
```

28.3.3　CAudioPlay 类的声明

下面的代码显示音频播放类的声明，其中声明在音频播放中用到的变量和成员函数：

```
01  class CAudioPlay : public IWMReaderCallback     //音频播放类定义
02  {
03      class CDataStack                            //数据堆栈类
04      {
05          char*    m_buffer;                      //缓冲区指针变量
06          long     m_length;                      //长度变量
07          CSync        m_sync;                    //同步对象变量
08      public:
09          CDataStack ();                          //构造函数
10          ~CDataStack ();                         //析构函数
11          void Append (const char* data, int len);//向缓冲区中添加
12          int Remove (char* data, int len);       //从缓冲区中移除
13          void  RemoveAll();                      //移除缓冲区中所有的数据
14          int Length ();                          //返回缓冲区中的数据长度
15      };
16  public:
17      static int WriteAudioThread (void* pThis);  //写入音频线程
18      int WriteAudioWait ();                      //等待写入音频
19      CSync        m_sync;                        //多线程同步
20      BOOL m_bPlayBack;                           //是否回放
21      CAudioPlay();                               //音频播放类的构造函数
22      CDlgItem*    pParDlg;                       //父窗口对象
23      //IUnknown 接口方法
24      HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid,
```

```
25         void __RPC_FAR *__RPC_FAR *ppvObject );
26       ULONG STDMETHODCALLTYPE AddRef();
27       ULONG STDMETHODCALLTYPE Release();
28       //IWMReaderCallback 接口方法，状态变化和采样
29       HRESULT STDMETHODCALLTYPE OnStatus(
30           /* [in] */ WMT_STATUS Status,
31           /* [in] */ HRESULT hr,
32           /* [in] */ WMT_ATTR_DATATYPE dwType,
33           /* [in] */ BYTE __RPC_FAR *pValue,
34           /* [in] */ void __RPC_FAR
35           *pvContext );
36       HRESULT STDMETHODCALLTYPE OnSample(
37           /* [in] */ DWORD dwOutputNum,
38           /* [in] */ QWORD cnsSampleTime,
39           /* [in] */ QWORD cnsSampleDuration,
40           /* [in] */ DWORD dwFlags,
41           /* [in] */ INSSBuffer __RPC_FAR *pSample,
42           /* [in] */ void __RPC_FAR *pvContext );
43       //CAudioPlay 方法
44       HRESULT Init();                              //初始化
45       HRESULT Exit();                              //退出
46       HRESULT Open( LPCWSTR pwszUrl, LPCWSTR pwszSaveUrlA,
47           LPCWSTR pwszSaveUrlB, LPCWSTR pwszSaveUrlC, BOOL bPlayBack);
48       //打开
49       HRESULT Close();                             //关闭
50       HRESULT Start( QWORD cnsStart = 0 );         //启动
51       HRESULT Stop();                              //停止
52       HRESULT Pause();                             //暂停
53       HRESULT Resume();                            //恢复
54   #ifdef SUPPORT_DRM
55       HRESULT ReopenReader( void *pvContext );     //重新打开读取进程
56   #endif
57       void SetAsyncEvent( HRESULT hrAsync );       //设置同步对象
58       QWORD GetFileDuration();                     //获取文件进度
59       BOOL IsSeekable();                           //是否可以拖动
60       BOOL IsBroadcast();                          //是否是广播
61       static DWORD WINAPI  OnWaveOutThread( LPVOID lpParameter );
62       //音频输出线程
63       static void CALLBACK WaveProc( HWAVEOUT hwo, UINT uMsg,
64           DWORD dwInstance, DWORD dwParam1,  DWORD dwParam2 );
65   private:
66       ~CAudioPlay();                               //析构函数
67       HRESULT GetHeaderAttribute( LPCWSTR pwszName,BYTE** ppbValue);
68       //获取头属性
69       HRESULT RetrieveAndDisplayAttributes();      //重新获取和显示属性
70       HRESULT GetAudioOutput();                    //获取音频输出
71       void WaitForEvent( HANDLE hEvent, DWORD msMaxWaitTime = INFINITE );
72                                                    //等待事件
73       void OnWaveOutMsg();                         //音频输出消息
74       void  OnWriteAudioMsg();                     //写入音频消息
75   #ifdef SUPPORT_DRM
76       BOOL m_bProcessingDRMOps;            //如果版权允许播放内容则返回 true
77   #endif
78       BOOL     m_bClosed;                  //如果内容打开了，则返回 true
79       BOOL     m_bIsSeekable;              //如果内容可以定位，则返回 true
80       BOOL     m_bIsBroadcast;             //如果播放广播流，则返回 true
```

```
81      BOOL  m bEOF;                            //如果播放到结尾，则返回 true
82      DWORD     m_dwThreadID;                  //音频输入线程
83      DWORD      m_dwAudioOutputNum;                //音频输出数目
84      HANDLE                m_hAsyncEvent;          //事件句柄
85      HRESULT               m_hrAsync;              //同步操作结果
86      IWMReader*            m_pReader;              //IWMReader 指针
87      IWMHeaderInfo*        m_pHeaderInfo;          //IWMHeaderInfo 指针
88      IWMReaderAdvanced4* m_pReaderAdvanced4;
89      IWMReaderNetworkConfig2*        m_pReaderNetworkConfig2;
90      LONG                  m cRef;                 //引用计数
91      LONG              m_cHeadersLeft;             //音频输出设备回放缓冲区
92      LPWSTR                m_pwszURL;              //URL
93      char              m_SaveURLA[MAX_PATH];
94      char              m_SaveURLB[MAX_PATH];
95      char              m_SaveURLC[MAX_PATH];
96      QWORD                 m_cnsFileDuration;      //内容的回放时间
97      WAVEFORMATEX*         m_pWfx;                 //音频格式结构
98      WAVEFORMATEX*         m_pSrcWfx;              //源音频格式
99      WAVEFORMATEX*         m_pDstWfx;              //目的音频格式
100     WAVEFORMATEX*         m_pMidWfx;              //中间音频格式
101     HACMDRIVERID      hadid;                      //ACM 驱动 ID
102     CDataStack            m_AudioData;            //音频数据
103     int                   nFileNo;                //文件号
104     int                   nFileByte;              //文件字节数
105     TCHAR             m_tszErrMsg[256];           //消息变量
106     HANDLE                hWriteThread;           //写线程句柄
107     DWORD                 dwWriteThreadID;        //写线程 ID
108 };
```

28.3.4　音频播放器初始化

音频播放器初始化由 CAudioPlay 类的 Init()函数完成。它实现有关音频播放器的初始化工作，包括定义所需变量、初始化运行环境、创建播放事件以及创建音频播放对象。代码如下：

```
01  HRESULT CAudioPlay::Init()
02  {
03      //Step1: 定义变量
04      HRESULT hr = S_OK;
05      //Step2: 初始化运行环境
06      do
07      {
08          //Step2.1: 初始化 COM 运行环境
09          hr = CoInitialize( NULL );
10          if( FAILED( hr ) )                  //判断处理结果，并输出错误信息
11          {
12              _tcscpy( m_tszErrMsg,
13                  T( "CoInitialize failed" ) );
14              pParDlg|InsertLog( m tszErrMsg );
15              break;
16          }
17          //Step2.2: 创建异步调用事件，当此类中的代码以异步方式调用
18          m_hAsyncEvent = CreateEvent( NULL,
19              false, false, NULL );
```

```
20          if( NULL == m_hAsyncEvent )      //判断处理结果，并输出错误信息
21          {
22              _tcscpy( m_tszErrMsg,
23                  _T( "Could not create the event" ) );
24              pParDlg|InsertLog( m_tszErrMsg );
25              hr = E_FAIL;
26              break;
27          }
28          //Step2.3: 创建读对象，只请求回放功能
29          hr = WMCreateReader( NULL,
30              WMT_RIGHT_PLAYBACK, &m_pReader );
31          if( FAILED( hr ) )                //判断处理结果，并输出错误信息
32          {
33              _tcscpy( m_tszErrMsg,
34                  _T( "Could not create Reader" ) );
35              pParDlg|InsertLog( m_tszErrMsg );
36              break;
37          }
38      }
39      while( false );
40      //Step3: 处理结果
41      if( FAILED( hr ) )                    //判断处理结果，并输出错误信息
42      {
43          wsprintf( m_tszErrMsg,
44              _T("%s (hr=%#X)"), m_tszErrMsg, hr );
45          pParDlg|InsertLog( m_tszErrMsg );
46          Exit();
47      }
48      return( hr );
49  }
```

28.3.5　音频采样处理

音频采样处理由 CAudioPlay 对象的 OnSample()函数完成。它实现音频样本数据的获取和音频样本数据的格式转换。代码如下：

```
01  //函数功能: IWMReaderCallback 方法，用于处理样本
02  HRESULT CAudioPlay::OnSample(
03      /* [in] */ DWORD dwOutputNum,
04      /* [in] */ QWORD cnsSampleTime,
05      /* [in] */ QWORD cnsSampleDuration,
06      /* [in] */ DWORD dwFlags,
07      /* [in] */ INSSBuffer __RPC_FAR *pSample,
08      /* [in] */ void __RPC_FAR *pvContext )
09  {
10      //定义变量
11      BYTE          *pData = NULL;
12      BYTE      *pDstMidData = NULL;
13      BYTE      *pDstData = NULL;
14      DWORD         cbData = 0;
15      DWORD         dwSrcBytes = 0 ;
16      DWORD         dwDstMidSamples = 0 ;
17      DWORD         dwDstMidBytes = 0 ;
18      DWORD         dwDstBytes = 0 ;
19      HRESULT     hr = S_OK;
20      MMRESULT    mmr = 0;
21      HACMDRIVER   had = NULL;
```

```
22      HACMSTREAM   hstrPcm = NULL;
23      HACMSTREAM   hstrDst = NULL;
24      ACMSTREAMHEADER astrhdrDst;
25      ACMSTREAMHEADER astrhdrPcm;
26      int          nMaxByte = 0;
27      //检查输出编号是否与存储的相符
28      //因为只存储第一个相符的音频输出，所以其他的输出，不管其类型是什么，都将忽略
29
30      if( dwOutputNum != m dwAudioOutputNum )     //判断处理结果
31      {
32          return( S OK );
33      }
34      //从 buffer 对象中获取样本数据
35      hr = pSample->GetBufferAndLength( &pData, &cbData );
36      if( FAILED( hr ) )                           //判断处理结果
37      {
38          return( hr );
39      }
40      //此方法内其余的代码，将使用 Windows Multimedia Wave 处理函数播放内容
41      //为数据头和数据分配内存，复制数据
42      do
43      {
44          //将源媒体转换为 CODEC 支持的 PCM 格式
45          //使用任何一个可以完成 PCM 到 PCM 转换的驱动
46          //打开转换流
47          mmr = acmStreamOpen(&hstrPcm,NULL, m_pSrcWfx,
48              m_pMidWfx, NULL,NULL,
49              0,ACM_STREAMOPENF_NONREALTIME);
50          //判断处理结果，并输出错误信息
51          if (mmr)
52          {
53              pParDlg->InsertLog(
54                  "Failed to open a stream to do PCM
55                  to PCM conversion\n");
56              hr = HRESULT FROM WIN32( GetLastError() ) ;
57              goto HappenError;
58              //为转换结果分配一个 buffer
59              dwSrcBytes = cbData ;
60              dwDstMidSamples =
61                  cbData  / m_pSrcWfx|wBitsPerSample *8 * m_pMidWfx
62                  |nSamplesPerSec / m_pSrcWfx|nSamplesPerSec;
63              dwDstMidBytes = dwDstMidSamples * m_pMidWfx
64                  |wBitsPerSample / 8;
65              pDstMidData = new BYTE [dwDstMidBytes];
66  #ifdef  DEBUG
67              memset(pDstMidData, 0, dwDstMidBytes);
68  #endif
69              //填充转换信息
70              memset(&astrhdrPcm, 0, sizeof(astrhdrPcm));
71              astrhdrPcm.cbStruct = sizeof(astrhdrPcm);
72              astrhdrPcm.pbSrc = pData;
73              astrhdrPcm.cbSrcLength = cbData;
74              astrhdrPcm.pbDst = pDstMidData;
75              astrhdrPcm.cbDstLength = dwDstMidBytes;
76              //准备转换头
77              mmr = acmStreamPrepareHeader(hstrPcm, &astrhdrPcm, 0);
78              if (mmr) //判断处理结果，并输出错误信息
79              {
80                  pParDlg->InsertLog("Failed acmStreamPrepareHeader\n");
```

```
81                  hr = HRESULT FROM WIN32( GetLastError() );
82                  goto HappenError;
83              }
84              //转换数据
85              mmr = acmStreamConvert(hstrPcm, &astrhdrPcm, 0);
86              if (mmr) //判断处理结果，并输出错误信息
87              {
88                  pParDlg->InsertLog("Failed to do PCM to
89                    PCM conversion\n");
90                  hr = HRESULT FROM WIN32( GetLastError() );
91                  goto HappenError;
92              }
93              //将数据从中间 PCM 格式转换为最终格式
94              //打开驱动
95              mmr = acmDriverOpen(&had, hadid, 0);
96              if (mmr) //判断处理结果，并输出错误信息
97              {
98                  pParDlg->InsertLog("Failed to open driver\n");
99                  hr = HRESULT_FROM_WIN32( GetLastError() );
100                 goto HappenError;
101             }
102             //打开转换流，必须使用 ACM_STREAMOPENF_NONREALTIME 标志
103             //如果不使用此标志，有些压缩软件将报告 512 号错误
104             mmr = acmStreamOpen(&hstrDst,    had,
105                 m pMidWfx,m pDstWfx, NULL, NULL,
106                 0,ACM STREAMOPENF NONREALTIME);
107             if (mmr) //判断处理结果，并输出错误信息
108             {
109                 pParDlg->InsertLog("打开流到 PCM 设备格式转换失败\n");
110                 hr = HRESULT FROM WIN32( GetLastError() );
111                 goto HappenError;
112             }
113             //为转换结果分配一个 buffer
114             //根据平均比特率加一些冗余，计算输出缓冲区的大小
115             //IMA_ADPCM 驱动在没有附加空间时，进行转换将失败
116             dwDstBytes = m_pDstWfx
117                 |nAvgBytesPerSec * dwDstMidSamples / m_pMidWfx
118                 |nSamplesPerSec;
119             dwDstBytes =  dwDstBytes;//* 3 / 2; //add a little room
120             pDstData = new BYTE [dwDstBytes];
121 #ifdef _DEBUG
122             memset(pDstData, 0, dwDstBytes);
123 #endif
124             //填充转换信息
125             memset(&astrhdrDst, 0, sizeof(astrhdrDst));
126             astrhdrDst.cbStruct = sizeof(astrhdrDst);
127             //the source data to convert
128             astrhdrDst.pbSrc = pDstMidData;
129             //dwDst1Bytes;
130             astrhdrDst.cbSrcLength = astrhdrPcm.cbDstLengthUsed ;
131             astrhdrDst.pbDst = pDstData;
132             astrhdrDst.cbDstLength = dwDstBytes;
133             //准备信息头
134             mmr = acmStreamPrepareHeader(hstrDst, &astrhdrDst, 0);
135             if (mmr) //判断处理结果，并输出错误信息
136             {
137                 pParDlg->InsertLog("Failed acmStreamPrepareHeader\n");
138                 hr = HRESULT_FROM_WIN32( GetLastError() );
139                 goto HappenError;
```

```
140             }
141             //转换数据
142             mmr = acmStreamConvert(hstrDst, &astrhdrDst, 0);
143             if (mmr) //判断处理结果，并输出错误信息
144             {
145                 pParDlg->InsertLog(
146                     "Failed to do PCM to driver format conversion\n");
147                 hr = HRESULT_FROM_WIN32( GetLastError() );
148                 goto HappenError;
149             }
150             //将转换后的数据写入文件
151             m_sync.Enter();
152             m_AudioData.Append((const char*)astrhdrDst.pbDst,
153                 astrhdrDst.cbDstLengthUsed);
154             m_sync.Leave();
155             mmr = acmStreamUnprepareHeader(hstrPcm,
156                 &astrhdrPcm, 0);
157             if (mmr) //判断处理结果，并输出错误信息
158             {
159                 pParDlg->InsertLog(
160                     "Failed to acmStreamUnprepareHeader\n");
161                 hr = HRESULT_FROM_WIN32( GetLastError() );
162                 goto HappenError;
163             }
164             //关闭转换流
165             acmStreamClose(hstrPcm, 0);
166             mmr = acmStreamUnprepareHeader(hstrDst,
167                 &astrhdrDst, 0);
168             if (mmr) //判断处理结果，并输出错误信息
169             {
170                 pParDlg->InsertLog(
171                     "Failed to acmStreamUnprepareHeader\n");
172                 hr = HRESULT_FROM_WIN32( GetLastError() );
173                 goto HappenError;
174             }
175             //关闭转换流和驱动
176             mmr = acmStreamClose(hstrDst, 0);
177             mmr = acmDriverClose(had, 0);
178             mmr = 0;
179             //判断是否需要回放，需要则回放数据
180             //设置播放的百分比
181             if( m_bIsBroadcast )
182             {
183                 pParDlg->SetTime( cnsSampleTime, 0 );
184             }
185             else
186             {
187                 pParDlg->SetTime( cnsSampleTime,
188                     m_cnsFileDuration );
189             }
190             SAFE_ARRAYDELETE( pDstMidData );//释放中间数据
191             SAFE_ARRAYDELETE( pDstData );    //释放数据
192         }
193         while( false );
194         //如果失败，则停止播放
195         if( MMSYSERR_NOERROR != mmr )    //判断处理结果，并输出错误信息
196         {
197             pParDlg->InsertLog(
198                 "Wave function failed" );
```

```
199             SendMessage((HWND)pParDlg,
200                 WM_COMMAND, IDC_STOP, 0);
201         }
202         SAFE_ARRAYDELETE( pDstMidData );     //释放中间数据
203         SAFE_ARRAYDELETE( pDstData );        //释放数据
204         return ( S_OK );                     //返回成功
205 HappenError:                                 //如果发生错误
206         mmr = acmStreamUnprepareHeader(
207             hstrPcm, &astrhdrPcm, 0);        //释放头信息
208         if (mmr)                         //判断处理结果，并输出错误信息
209         {
210             pParDlg->InsertLog("Failed to acmStreamUnprepareHeader\n");
211             hr = HRESULT_FROM_WIN32( GetLastError() );
212             goto HappenError;
213         }
214         //关闭转换流
215         acmStreamClose(hstrPcm, 0);
216         mmr = acmStreamUnprepareHeader(
217             hstrDst, &astrhdrDst, 0);
218         if (mmr)                         //判断处理结果，并输出错误信息
219         {
220             pParDlg->InsertLog(
221                 "Failed to acmStreamUnprepareHeader\n");
222             hr = HRESULT_FROM_WIN32( GetLastError() );
223             goto HappenError;
224         }
225         //关闭转换流和驱动
226         mmr = acmStreamClose(hstrDst, 0);
227         mmr = acmDriverClose(had, 0);
228         mmr = 0;
229         SAFE_ARRAYDELETE( pDstMidData );     //释放中间数据
230         SAFE_ARRAYDELETE( pDstData );        //释放数据
231         return( S_OK );                      //返回成功
232     }
233 }
```

28.3.6　音频输出实现

音频输出由 CAudioPlay 对象的 OnWaveOutMsg()函数完成。它实现创建消息队列以及循环处理消息队列中的 WOM_DONE 消息的功能。代码如下：

```
01  void CAudioPlay::OnWaveOutMsg()         //函数功能：WaveOut 消息处理
02  {
03      //Step1：定义变量
04      HRESULT     hr = S_OK;
05      LPWAVEHDR   pwh = NULL;
06      MMRESULT    mmr = MMSYSERR_NOERROR;
07      MSG         uMsg;
08      //Step2：创建消息队列
09      PeekMessage( &uMsg, NULL, WM_USER, WM_USER, PM_NOREMOVE );
10      //Step3：消息队列已经创建了，进行读取消息操作
11      while( 0 != GetMessage( &uMsg, NULL, 0, 0 ) )
12      {
13          switch( uMsg.message )
14          {
15          case WOM_DONE:
```

```
16              //Step3.1：重置 wave header，因为已经播放
17              pwh = ( LPWAVEHDR )uMsg.wParam;
18              mmr = waveOutUnprepareHeader(
19                  m_hWaveOut, pwh,sizeof( WAVEHDR ) );
20              if( MMSYSERR_NOERROR == mmr )//判断处理结果，并输出错误信息
21              {
22                  InterlockedDecrement( &m_cHeadersLeft );
23              }
24              else if( WHDR_ENDLOOP == mmr )
25              {
26                  SetEvent( m_hAsyncEvent );    //设置事件
27              }
28              else
29              {
30                  //发生错误时停止播放
31                  SendMessage((HWND)pParDlg,
32                      WM_COMMAND, IDC_STOP, 0);
33                  pParDlg->InsertLog(
34                      "Wave function (waveOutUnprepareHeader) failed" );
35              }
36              //Step3.2：如果没有空闲的缓冲区则停止
37              if( m_bEOF && ( 0 == m_cHeadersLeft ) )
38              {
39                  pParDlg->SetCurrentStatus( STOP );
40              }
41              SAFE_ARRAYDELETE(pwh);            //释放 pwh
42              break;
43          case WOM_CLOSE:
44              PostQuitMessage( 0 );             //函数退出
45              break;
46          }
47      }
48      return;
49  }
```

28.3.7　打开音频文件

打开音频文件由 CAudioPlay 对象的 Open()函数完成。它实现打开网络音频文件、获取并显示音频属性以及获取音频输出的功能，为音频播放做好准备工作。代码如下：

```
01  HRESULT CAudioPlay::Open( LPCWSTR pwszUrl,
02      LPCWSTR pwszSaveUrlA, LPCWSTR pwszSaveUrlB,
03      LPCWSTR pwszSaveUrlC, BOOL bPlayBack)          //打开音频
04  {
05      //Step1：定义变量
06      const INT MAX_ERROR_LENGTH = 256;
07      HRESULT hr = S_OK;
08      //Step2：检查输入参数是否不为 NULL，并且 reader 已经初始化
09      if( NULL == pwszUrl )
10          return( E_INVALIDARG );
11      if( NULL == pwszSaveUrlA )
12          return( E_INVALIDARG );
13      if( NULL == pwszSaveUrlB )
14          return( E_INVALIDARG );
15      if( NULL == m_pReader )
16          return( E_UNEXPECTED );
17      do                                             //执行 do 循环
```

```
18      {
19          ZeroMemory( m_tszErrMsg, MAX_ERROR_LENGTH );
20          ResetEvent( m_hAsyncEvent );
21          //Step2.1：关闭以前打开的文件，并删除旧文件
22          Close();
23          SAFE_ARRAYDELETE( m_pwszURL );              //删除释放变量
24          //Step2.2：设置新文件名
25          m_pwszURL = new WCHAR[ wcslen( pwszUrl ) + 1 ];
26          if( NULL == m_pwszURL )  //判断处理结果，并输出错误信息
27          {
28              hr = HRESULT_FROM_WIN32( GetLastError() );
29              _tcscpy( m_tszErrMsg, _T( "Insufficient memory" ) );
30              pParDlg->InsertLog( m_tszErrMsg );
31              break;
32          }
33          //初始化各个变量值
34          wcscpy( m_pwszURL, pwszUrl );
35          memset(m_SaveURLA, 0x00, sizeof(m_SaveURLA));
36          memset(m_SaveURLB, 0x00, sizeof(m_SaveURLB));
37          memset(m_SaveURLC, 0x00, sizeof(m_SaveURLC));
38          memcpy(m_SaveURLA, pwszSaveUrlA, wcslen( pwszSaveUrlA ));
39          memcpy(m_SaveURLB, pwszSaveUrlB, wcslen( pwszSaveUrlB ));
40          memcpy(m_SaveURLC, pwszSaveUrlC, wcslen( pwszSaveUrlC ));
41          remove(m_SaveURLA);
42          remove(m_SaveURLB);
43          remove(m_SaveURLC);
44  #ifdef SUPPORT_DRM
45          //初始化 DRM 对象
46          hr = objDRM.Init( this, m_pReader, m_pwszURL );
47          if( FAILED( hr ) )                //判断处理结果，并输出错误信息
48          {
49              break;
50          }
51          m_bProcessingDRMOps = false;      //赋值进程处理变量为 false
52  #endif
53          //Step2.3：用 reader 对象打开文件。此方法同时设置 reader 的状态
54          hr = m_pReader->Open( pwszUrl, this, NULL );
55          if( FAILED( hr ) )                //判断处理结果，并输出错误信息
56          {
57               tcscpy( m_tszErrMsg,
58                  T( "Could not open the specified file" ) );
59              break;
60          }
61          //Step2.4：等待打开调用结束。当 reader 完成时，会调用 OnStatus()回调函数
62
63          WaitForEvent( m_hAsyncEvent );
64  #ifdef SUPPORT_DRM
65          if( ( NS_S_DRM_ACQUIRE_CANCELLED == m_hrAsync ) ||
66              ( NS_E_DRM_INDIVIDUALIZATION_INCOMPLETE == m_hrAsync ) )
67          {
68              //判断处理结果
69              //DRM 操作已经被取消，因此文件还未打开
70              hr = m_hrAsync;
71              return( hr );
72          }
73          //Step2.5：查询许可失败的原因
74          BOOL fAcquiringLicenseNonSilently = false;
75          if( 7 == objDRM.GetLastDRMVersion() &&
76              ( NS_E_DRM_LICENSE_NOTACQUIRED == m_hrAsync ||
```

```
77                  NS E DRM LICENSE STORE ERROR == m hrAsync ) )
78              {
79                  fAcquiringLicenseNonSilently = true;
80                  hr = objDRM.AcquireLastV7LicenseNonSilently();
81                  if ( FAILED( hr ) )                     //判断处理结果
82                  {
83                      return hr;
84                  }
85              }
86              if( NS_E_DRM_NO_RIGHTS == m_hrAsync ||
87                  fAcquiringLicenseNonSilently )
88              {
89                  //判断处理结果
90                  MessageBox( NULL,
91                       T( "After acquiring the license,re-open the file."),
92                      ERROR DIALOG TITLE, MB OK );
93                  SetCurrentStatus( CLOSED );             //设置状态为关闭
94                  SetCurrentStatus( READY );              //设置状态为准备好
95                  hr = m hrAsync;
96                  return( hr );                           //返回句柄
97              }
98      #endif
99              if( FAILED( m_hrAsync ) )                   //判断处理结果
100             {
101                 hr = m hrAsync;
102                  tcscpy( m tszErrMsg,
103                     T( "Could not open the specified file" ) );
104                 break;
105             }
106             SAFE_RELEASE( m_pHeaderInfo );              //安全释放头信息指针
107             //查询头信息
108             hr = m pReader->QueryInterface( IID IWMHeaderInfo,
109                 ( VOID ** )&m_pHeaderInfo );
110             if( FAILED (hr) )                           //判断处理结果
111             {
112                 _tcscpy( m_tszErrMsg,
113                     _T( "Could not QI for IWMHeaderInfo" ) );
114                 break;
115             }
116             hr = RetrieveAndDisplayAttributes();//获取并显示属性
117             if( FAILED( hr ) )                          //判断处理结果
118             {
119                 break;
120             }
121             hr = GetAudioOutput();                      //获取音频输出
122             if( FAILED( hr ) )                          //判断处理结果
123             {
124                 break;
125             }
126         }
127         while( false );
128         if( FAILED( hr ) )                              //判断处理结果
129         {
130             Close();                                    //关闭
131             if( _tcslen( m_tszErrMsg ) > 0 )            //输出错误信息
132             {
133                 wsprintf( m tszErrMsg,
134                     T("%s (hr=%#X)"), m tszErrMsg, hr );
135             }
```

```
136     }
137     m_bPlayBack = bPlayBack;                    //赋值回放变量
138     return( hr );                               //返回句柄
139 }
```

28.3.8　停止音频播放

停止音频播放由 CAudioPlay 对象的 Stop()函数完成。它实现停止获取音频输出以及重置音频播放器的各个状态值的功能。代码如下：

```
01  HRESULT CAudioPlay::Stop()                      //音频对象停止按钮处理函数
02  {
03      HRESULT hr = S_OK;                          //定义句柄对象
04      if( NULL == m_pReader )                     //判断读对象
05      {
06          return( E_UNEXPECTED );                 //返回异常
07      }
08      Sleep (10);
09      WaitForSingleObject (hWriteThread, 5000);//等待写线程
10      DWORD dwExitCode;
11      GetExitCodeThread (hWriteThread, &dwExitCode);//获取线程退出状态
12      if (dwExitCode == STILL_ACTIVE)             //如果线程依然进行
13      {
14          TerminateThread (hWriteThread, 0);  //终止线程
15      }
16      hWriteThread = NULL;                        //赋值写线程为 NULL
17  #ifdef SUPPORT_DRM
18      hr = objDRM.Cancel();                       //DRM 对象取消
19      if( FAILED( hr ) )                          //判断处理结果
20      {
21          return( hr );
22      }
23      if( S_OK == hr )                            //判断处理结果，如果成功
24      {
25          SetCurrentStatus( CLOSED );             //设置状态为关闭
26          SetCurrentStatus( READY );              //设置状态为准备好
27          return( hr );                           //返回句柄
28      }
29  #endif
30      hr = m_pReader->Stop();                     //停止读取
31      if( FAILED( hr ) )                          //判断处理结果
32      {
33          return( hr );
34      }
35      //重置音频输入设备
36      if( NULL != m_hWaveOut )
37      {
38          //重置音频输出
39          if( MMSYSERR_NOERROR != waveOutReset( m_hWaveOut ) )
40          {
41              return( E_FAIL );
```

```
42          }
43          WaitForEvent( m_hAsyncEvent );       //等待事件
44      }
45      pParDlg->SetCurrentStatus( CLOSED );     //设置状态为关闭
46      pParDlg->SetCurrentStatus( READY );      //设置状态为准备好
47      return( S_OK );                          //返回成功
48  }
```

28.3.9　暂停音频和继续音频

暂停音频播放由 CAudioPlay 对象的 Pause()函数完成。它实现暂停音频输出的功能。代码如下：

```
01  HRESULT CAudioPlay::Pause()                 //暂停处理函数
02  {
03       if( NULL == m_pReader )                //判断读对象是否有效
04       {
05          return( E_UNEXPECTED );
06       }
07      return( m_pReader->Pause() );           //暂停读取
08  }
```

继续音频播放由 CAudioPlay 对象的 Resume()函数完成。它实现从上次暂停音频输出点继续播放音频的功能。代码如下：

```
01  HRESULT CAudioPlay::Resume()                //恢复播放处理函数
02  {
03      if( NULL == m_pReader )                 //判断读对象是否有效
04      {
05          return( E_UNEXPECTED );
06      }
07      return( m_pReader->Resume() );          //恢复读取
08  }
```

28.3.10　获取音频属性

获取音频属性由 CAudioPlay 对象的 RetrieveAndDisplayAttributes()函数完成。它实现获取音频标题、音频作者、音频版权信息和音频播放进度等各种与音频相关的信息的功能。代码如下：

```
01  HRESULT CAudioPlay::RetrieveAndDisplayAttributes()//获取并显示属性
02  {
03      BYTE*   pbValue = NULL;                  //定义字节指针变量
04      HRESULT hr = S_OK;                       //定义操作结果句柄
05      WCHAR   wszNoData[] = L"No Data";        //初始化没有数据的提示
06      do                                       //循环获取
07      {
08          hr = GetHeaderAttribute( g_wszWMTitle, &pbValue );
09          //获取 Title 属性
10          if( FAILED( hr ) )                   //判断处理结果
```

```
11          {
12               tcscpy( m tszErrMsg,
13                T( "Could not get the Title attribute" ) );
14              break;
15          }
16          if( NULL != pbValue )                 //判断获取的值
17          {
18              pParDlg->SetItemText( IDC CLIP, ( LPWSTR )pbValue );
19              //设置获取的值
20              SAFE_ARRAYDELETE( pbValue );      //安全释放变量值
21          }
22          else
23          {
24              pParDlg->SetItemText( IDC CLIP, wszNoData );
25          }
26          hr = GetHeaderAttribute( g wszWMAuthor, &pbValue);
27          if( FAILED( hr ) )                    //判断处理结果
28          {
29              _tcscpy( m_tszErrMsg, _T( "不能获取作者的名称" ) );
30              break;
31          }
32          if( NULL != pbValue )                 //如果获取了有效值
33          {
34              pParDlg->SetItemText( IDC AUTHOR, ( LPWSTR )pbValue );
35              //显示作者信息
36              SAFE_ARRAYDELETE( pbValue );      //安全释放变量
37          }
38          else
39          {
40              pParDlg->SetItemText( IDC AUTHOR, wszNoData );
41              //设置作者名称为默认值
42          }
43          hr = GetHeaderAttribute( g wszWMCopyright, &pbValue );
44          //获取 Copyright 属性
45          if( FAILED( hr ) )                    //判断处理结果
46          {
47              _tcscpy( m_tszErrMsg, _T( "不能获取版权信息" ) );
48              break;
49          }
50          if( NULL != pbValue )                 //如果获取了有效值
51          {
52              pParDlg->SetItemText( IDC COPYRIGHT, ( LPWSTR )pbValue );
53              //显示版权信息
54              SAFE ARRAYDELETE( pbValue );      //安全释放变量
55          }
56          else
57          {
58              pParDlg->SetItemText( IDC COPYRIGHT, wszNoData );
59              //设置版权为默认值
60          }
61          hr = GetHeaderAttribute( g wszWMDuration, &pbValue );
62          //获取 Duration 属性
```

```
63          if( FAILED( hr ) )                      //判断处理结果
64          {
65              _tcscpy( m_tszErrMsg, _T( "不能获取进度信息" ) );
66              break;
67          }
68          if( NULL != pbValue )                   //如果获取了有效值
69          {
70              m_cnsFileDuration = *( QWORD* )pbValue;
71              SAFE_ARRAYDELETE( pbValue );        //安全释放变量
72          }
73          else
74          {
75              m_cnsFileDuration = 0;              //设置进度为文件头默认值
76          }
77          pParDlg->SetTime( 0, m_cnsFileDuration );//设置进度
78          hr = GetHeaderAttribute( g wszWMSeekable, &pbValue );
79          //获取 Seekable 属性
80          if( FAILED( hr ) )                      //判断处理结果
81          {
82              _tcscpy( m_tszErrMsg, _T( "不能获取是否可以拖动" ) );
83              break;
84          }
85          if( NULL != pbValue )                   //如果获取了有效值
86          {
87              m_bIsSeekable = *( BOOL* )pbValue;
88          }
89          else
90          {
91              m bIsSeekable = false;
92          }
93          hr = GetHeaderAttribute( g_wszWMBroadcast, &pbValue );
94          //获取 Broadcast 属性
95          if( FAILED( hr ) )
96          {
97              _tcscpy( m_tszErrMsg, _T( "不能获取广播属性" ) );
98              break;
99          }
100         If( NULL != pbValue )                   //如果获取了有效值
101         {
102             m_bIsBroadcast = *( BOOL* )pbValue;
103         }
104         else
105         {
106             m bIsBroadcast = false;
107         }
108     }
109     while (false );
110     if( FAILED( hr ) )                          //判断返回的句柄，并输出信息
111     {
112         wsprintf( m_tszErrMsg, _T("%s (hr=%#X)"), m_tszErrMsg, hr );
113     }
114     return( hr );                               //返回句柄
115 }
```

28.4　程序运行效果

编写完程序代码，编译并链接程序后，调试运行程序，其运行效果如图 28-2 所示。在数据源和目的文件中输入相应的信息，并单击“开始转换”按钮，程序就会缓冲网络电台资源，并可以选择进行回放，同时会在信息选项卡中显示当前缓冲的音频的剪辑、作者和版权等信息。

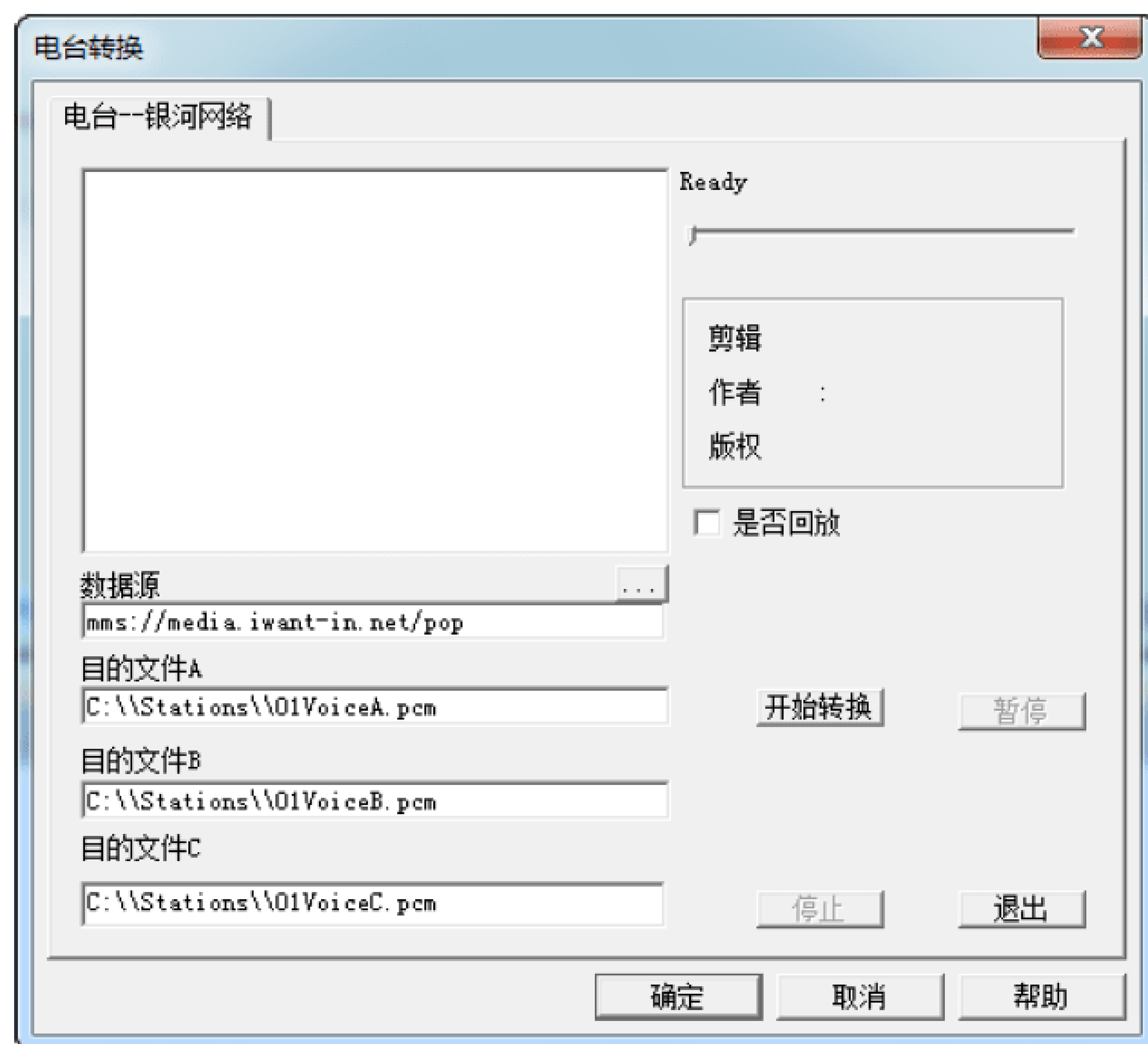


图 28-2　电台音频播放程序运行效果

28.5　本 章 小 结

本章使用 VC 编写了一个网络电台的音频播放程序。本章重点讲解了 Windows 标准控件的使用、多线程的实现、音频驱动函数的使用以及 COM 技术的使用。在学习本章时除了学习这些知识的使用，同时需要学习编程思想，如进行错误处理、编写高效安全的多线程程序等各个方面。第 29 章将介绍应用较广的 GPS 定位系统的开发。

第 29 章　GPS 定位系统

GPS（Global Position System，全球定位系统）由美国从 20 世纪 70 年代开始研制，历经 20 年研制成功。虽然最初是为军用而设计的，现在它已经非常成熟地应用于民用的各个行业。随着人们对许多行业的专业化要求越来越高，应用范围也越来越广。本章中将会涉及有关网络编程方面的知识。

29.1　GPS 监控系统概况

GPS 监控系统是 GPS 的典型应用，可以实现位置的确定、轨迹的追踪、线路的导航等功能。读者可以开发出符合自己需求的 GPS 监控系统。本节主要介绍关于 GPS 监控系统的概况。

29.1.1　GPS 监控系统概述

GPS 系统可以实现测绘勘察和定位导航等功能。广泛应用于交通行业、物流行业、公安系统和海洋测绘等各个行业。因为应用范围非常广，所以，系统结构也千差万别。

从 GPS 系统结构来看，GPS 系统主要分为两类，一类是自助式 GPS 系统，此类系统由 GPS 模块通过标准接口（串口、USB 接口和蓝牙等）直接连接到计算机和 PocketPC 等设备中，并根据用户的需求编写合适的应用程序，由用户自助使用，如自助车载导航系统、GPS 轨迹记录仪等设备。另一类是中心式 GPS 监控系统，此类系统主要是由终端设备（包含 GPS 模块和无线通信模块）和监控中心组成的一个服务网络，监控中心通过无线通信模块与终端设备进行数据通信，从而实现对终端设备的监视和控制。因为本章主要介绍串口的编程知识，所以选择自助式 GPS 监控系统。

GPS 监控系统的主要功能是实现对终端设备的监控。GPS 监控系统从终端设备中接收 GPS 信息，将其解析，在监控中心的地图上显示其位置，并可以实现其历史轨迹的回放。

一般的 GPS 监控系统由远程终端设备、监控中心软件和数据传输通道组成，因此，GPS 监控系统结合了 GPS（Global Position Systems，全球定位系统）技术、GIS（Global Information Systems，地理信息系统）技术和 GSM 技术。其中，GPS 技术负责处理设备终端的定位，GIS 负责处理有关地理信息，也就是地图的知识，GSM 技术负责数据通道，因此 GPS 监控系统也称为 3G 系统。当然，随着技术的发展，GSM 技术也可以由 GPRS 或 CDMA 技术代替。

29.1.2　GPS 监控系统的系统架构

GPS 监控系统主要由通信前置机、中心服务器、数据库服务器、Web 服务器和监控终

端组成，如图 29-1 所示为 GPS 监控系统的系统架构图。

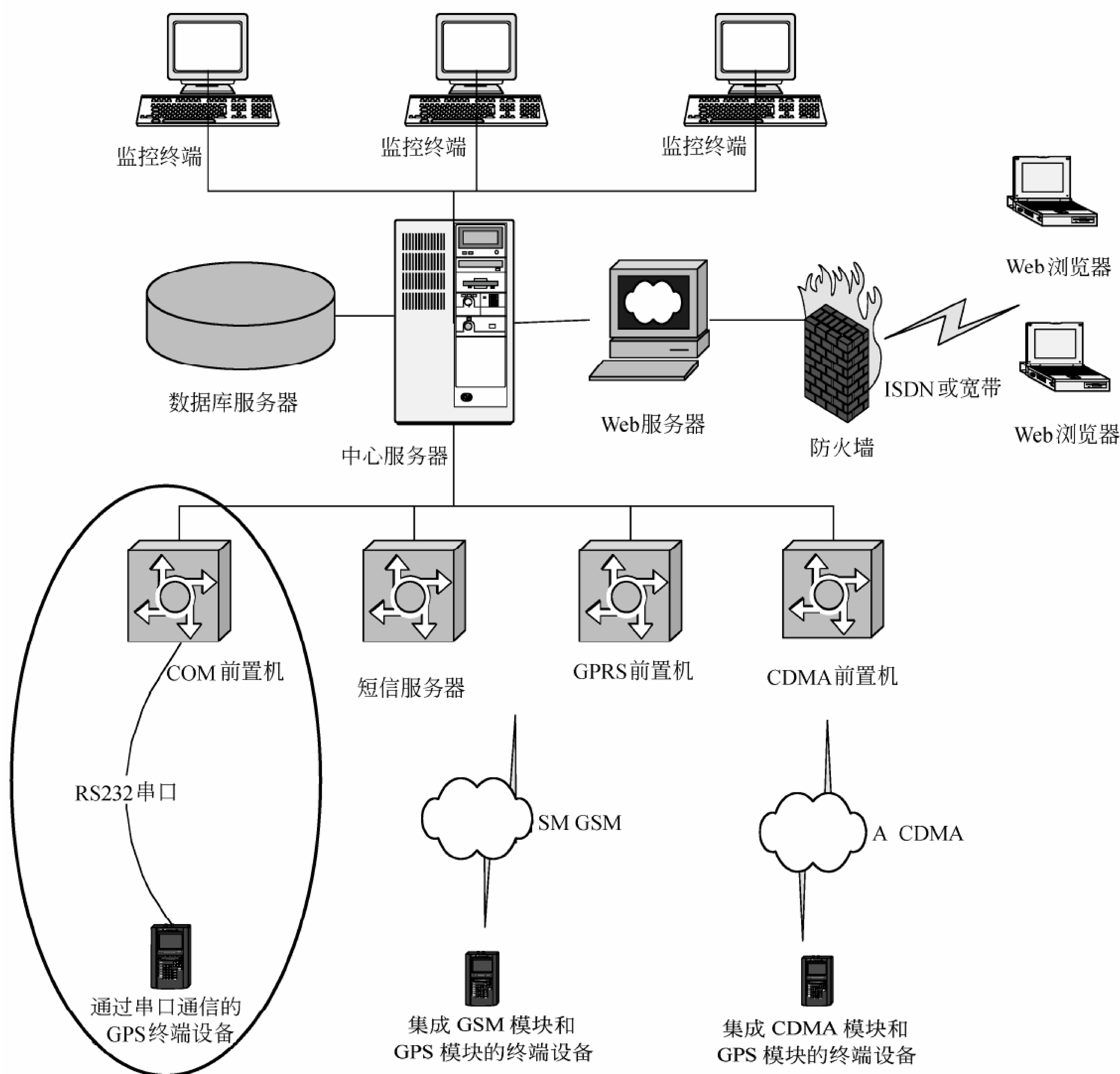


图 29-1　GPS 监控系统的系统架构图

1．通信前置机

在了解通信前置机之前，首先需要了解终端设备与中心系统之间的通信方式。目前有 3 种，分别是短信方式、无线数据网络方式和串口方式。

- 短信方式：指在终端设备和中心系统之间通过短消息内容进行数据和命令的传输，这种方式可靠、速度比较快，但是通信费用比较高，所以一般作为下行命令的数据通信方式。
- 无线数据网络方式：是指通过 GPRS（中国移动通信）和 CDMA（中国联通）来进行数据和命令的传输，这种方式相对比较稳定，资费比较低，是目前的主流 GPS 终端设备和系统中心之间的通信方式。
- 串口方式：是指 GPS 模块通过串口直接与中心系统进行数据和命令的传输，严格地讲，这种接入方式不属于 GPS 监控系统的范围，因为串口的通信距离是有限的。

所以，GPS 模块总是与中心系统处在相同位置。但是可以通过简单的编程，实现对现有位置的定位，再结合 GIS 的编程，可以将其轨迹记录。本章就以此种方式为例。

所以，通信前置机包括短信前置机、GPRS 前置机、CDMA 前置机和 COM 前置机 4 种。

- 短信前置机支持集成了短信模块（GSM 模块或 CDMA 模块）和 GPS 模块，并使用短信作为通信方式的终端设备。
- GPRS 前置机支持集成了 GPRS 模块和 GPS 模块，并使用 GPRS 作为通信方式的终端设备。
- CDMA 前置机支持集成了 CDMA 模块和 GPS 模块，并使用 CDMA 作为通信方式的终端设备。
- COM 前置机支持直接通过 COM 串口连接计算机进行通信的 GPS 模块（此处也就是 GPS 接收机）。

其中，短信前置机又分为移动短信专线、联通短信专线和串口短信猫 3 种。

- 移动短信专线是通过专线直接连接到移动短消息服务中心进行短消息的收发。
- 联通短信专线是通过专线直接连接到联通短消息服务中心进行短消息的收发。
- 串口短信猫则将短信模块通过串口连接计算机，使其与终端设备之间进行短消息的收发，即点对点的短消息。但是前置机可以通过多串口卡等设备同时连接多个串口，以减轻前置机的压力。

一般情况下，当采用短信的方式与终端设备进行通信时，如果终端设备比较多，建议采用专线方式进行数据通信；如果终端设备比较少，建议采用短信猫的方式进行数据通信。

2．数据库服务器

数据库服务器用于存储整个系统中的数据，包括接收到的终端设备的位置信息、下发给终端设备的命令以及用户的操作记录等。数据库服务器通过中心服务器为其他模块服务。

3．监控终端

监控终端主要实现实际监控功能，包括位置查询和命令的下发等。主要的技术是位置信息与 GIS（地理信息系统）的结合，可以在地图上显示设备终端的实际位置，并可对其轨迹实现回放，这样用户可以直观地对终端设备进行监控。通过中心服务器与数据库服务器和前置机之间进行通信。

Web 服务器为系统提供 Web 使用方式，包括查看终端设备信息、向终端设备发送命令等功能。通过中心服务器与数据库服务器、前置机进行通信。

4．中心服务器

中心服务器是连接各个模块的核心部分，接收来自前置机的数据，并将其进行业务处理，而后存入数据库；同时，接收来自 Web 服务器和监控终端的命令，将命令进行业务处理，并存储数据库，发送到前置机中。因此，系统之间的各个部分是通过中心服务器进行通信的。

在实际的系统中，会根据实际情况，调整架构的某个部分。如果不为用户提供 Web 监控的功能，则 Web 服务器组件可以删除；前置机也会根据用户采取的通信方式来确定使用哪个或哪几个前置机。

在本章中，以 COM 前置机为例，主要讲述如何通过串口进行 GPS 数据的通信，如图 29-1 所示。此处，终端设备通过串口直接与计算机进行通信。实际应用中，终端设备与中

心系统之间除了传输 GPS 信息外，还会根据用户的需求，在每次通信会话中，加入与用户应用相关的业务数据。为了突出本章的重点，在本章的实例中，没有业务数据，终端设备以 GPS 接收机为例，将接收到的数据未经任何处理地通过串口传输给计算机；同时，中心系统这端也会简化，将其接收到的 GPS 信息解码后，在程序界面上显示出来。在实际应用中，应该将其发送给中心服务器。用户在理解了 GPS 监控系统的工作原理后，可以根据需求，修改扩展应用。

29.2　GPS 数据通信协议 NEMA0183 协议

有关 GPS 的数据通信协议有多种，但是目前市面上最常用的协议是 NEMA0183 协议。大部分 GPS 接收机和导航仪都符合该协议。该协议是由 NEMA（Nation Marine Electronics Association，美国国家海事电子协会）制定的关于 GPS 接收机的通信接口。本节主要介绍其协议格式。

29.2.1　配置参数及协议格式

NEMA0183 协议分为文本格式和 ASCII 格式。其中 ASCII 格式是以语句的形式定义的，其又分为 NEMA0183 标准语句和厂商扩展语句。NEMA0183 标准语句定义了 GPS 数据的常用格式，一般厂商都会遵循此标准。厂商扩展语句则是 NEMA 为各个厂商提供的扩展接口。各个厂商可以根据自己的需求扩展其 NEMA 语句。比较典型的是 GARMIN（美国高明）NEMA0183 扩展，根据 NEMA0183 扩展语句标准，定义了其 GPS 通信数据格式的企业标准，并且在其地图支持组件 MapSource 中只支持自定义语句。

NEMA0183 的通用协议语句格式为：

```
$aaaaa,df1,df2,… *hh0x0D0x0A
```

所有的协议都由$开始，以回车换行（0x0D0x0A）结束，参数之间以逗号分隔。其中 aaaaa 表示协议的语句名称，具体确定语句的含义。df1、df2 表示语句的参数，*表示校验码的开始，hh 表示语句的校验码，校验码是从$开始，直到*之间的数据进行异或的结果值。

虽然 NEMA0183 规定了 GPS 数据的标准格式，但是实际应用中，读者在进行 GPS 数据解析时，应该根据自己使用的芯片所使用的语句格式进行解析。

29.2.2　NEMA0183 标准语句

NEMA0183 协议定义了一组 NEMA0183 标准语句，这组语句中包含了常用的 GPS 语句。主要用于传输 GPS 定位信息、卫星信息、速度、时间和坐标系等常用的信息。NEMA0183 标准语句以 GP 开头，紧随其后的 3 位表示语句类型。常用的语句如表 29-1 所示。

表 29-1　NEMA0183 标准语句

位　　置	数据项含义	数据项取值说明
GPS 定位信息（GGA） $GPGGA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,M,<10>,M,<11>,<12>*hh<CR><LF>		
<1>	UTC 时间	hhmmss（时分秒）格式，前面的 0 也将被传输
<2>	纬度	ddmm.mmmm（度分）格式，前面的 0 也将被传输
<3>	纬度半球	N（北半球）或 S（南半球）
<4>	经度	dddmm.mmmm（度分）格式，前面的 0 也将被传输
<5>	经度半球	E（东经）或 W（西经）
<6>	GPS 状态	0=未定位，1=非差分定位，2=差分定位，6=正在估算
<7>	正在使用解算位置的卫星数量	（00～12），前面的 0 也将被传输
<8>	HDOP 水平精度因子	（0.5～99.9）
<9>	海拔高度	（–9999.9～99999.9）
<10>	地球椭球面相对大地水准面的高度	单位是 m
<11>	差分时间	从最近一次接收到差分信号开始的秒数，如果不是差分定位则为空
<12>	差分站 ID 号	0000～1023，前面的 0 也将被传输，如果不是差分定位则为空
当前卫星信息（GSA） $GPGSA,<1>,<2>,<3>,<3>,<3>,<3>,<3>,<3>,<3>,<3>,<3>,<3>,<3>,<3>,<4>,<5>,<6>*hh<CR><LF>		
<1>	模式	M=手动，A=自动
<2>	定位类型	1=没有定位，2=2D 定位，3=3D 定位
<3>	PRN 码（伪随机噪声码），正在用于解算位置的卫星号	（01～32），前面的 0 也将被传输
<4>	PDOP 位置精度因子	（0.5～99.9）
<5>	HDOP 水平精度因子	（0.5～99.9）
<6>	VDOP 垂直精度因子	（0.5～99.9）
可见卫星信息（GSV） $GPGSV,<1>,<2>,<3>,<4>,<5>,<6>,<7>,…<4>,<5>,<6>,<7>*hh<CR><LF> 注：<4>,<5>,<6>,<7>信息将按照每颗卫星进行循环显示，每条 GSV 语句最多可以显示 4 颗卫星的信息。其他卫星信息将在下一序列的 NEMA0183 语句中输出		
<1>	GSV 语句的总数	整型
<2>	本句 GSV 的编号	整型
<3>	可见卫星的总数	（00～12），前面的 0 也将被传输
<4>	PRN 码（伪随机噪声码）	（01～32），前面的 0 也将被传输
<5>	卫星仰角	（00～90°），前面的 0 也将被传输
<6>	卫星方位角	（000～359°），前面的 0 也将被传输
<7>	信噪比	（00～99dB），没有跟踪到卫星时为空，前面的 0 也将被传输
推荐定位信息（RMC） $GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>*hh<CR><LF>		
<1>	UTC 时间	hhmmss（时分秒）格式
<2>	定位状态	A=有效定位，V=无效定位
<3>	纬度	ddmm.mmmm（度分）格式，前面的 0 也将被传输

续表

位　　置	数据项含义	数据项取值说明
<4>	纬度半球	N（北半球）或 S（南半球）
<5>	经度	dddmm.mmmm（度分）格式，前面的 0 也将被传输
<6>	经度半球	E（东经）或 W（西经）
<7>	地面速率	（000.0～999.9kn），前面的 0 也将被传输。1kn（节）=0.51444m/s
<8>	地面航向	（000.0～359.9°），以正北为参考基准，前面的 0 也将被传输
<9>,	UTC 日期	ddmmyy（日月年）格式
<10>	磁偏角	（000.0～180.0°），前面的 0 也将被传输
<11>,	磁偏角方向	E（东）或 W（西）
<12>	模式指示	仅 NEMA0183 3.00 版本输出，A=自主定位，D=差分，E=估算，N=数据无效
地面速度信息（VTG） $GPVTG,<1>,T,<2>,M,<3>,N,<4>,K,<5>*hh<CR><LF>		
<1>	以真北为参考基准的地面航向	（000～359°），前面的 0 也将被传输
<2>	以磁北为参考基准的地面航向	（000～359°），前面的 0 也将被传输
<3>	地面速率	（000.0～999.9 kn），前面的 0 也将被传输
<4>	地面速率	（0000.0～1851.8 km/h），前面的 0 也将被传输
<5>	模式指示	仅 NEMA0183 3.00 版本输出，A=自主定位，D=差分，E=估算，N=数据无效
定位地理信息（GLL） $GPGLL,<1>,<2>,<3>,<4>,<5>,<6>,<7>*hh<CR><LF>		
<1>	纬度	ddmm.mmmm（度分）格式，前面的 0 也将被传输
<2>	纬度半球	N（北半球）或 S（南半球）
<3>	经度	dddmm.mmmm（度分）格式，前面的 0 也将被传输
<4>	经度半球	E（东经）或 W（西经）
<5>	UTC 时间	hhmmss（时分秒）格式
<6>	定位状态	A=有效定位，V=无效定位
<7>	模式指示	仅 NEMA0183 3.00 版本输出，A=自主定位，D=差分，E=估算，N=数据无效
时间和日期信息（ZDA） $GPZDA,<1>,<2>,<3>,<4>*hh<CR><LF>		
<1>	UTC 时间	hhmmss（时分秒）格式
<2>	UTC 日期，日	
<3>	UTC 日期，月	
<4>	UTC 日期，年	
大地坐标系信息（DTM） $GPDTML,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>*hh<CR><LF>		
<1>	本地坐标系代码	W84
<2>	坐标系子代码	空

续表

位　　置	数据项含义	数据项取值说明
<3>	纬度偏移量	
<4>	纬度半球	N（北半球）或 S（南半球）
<5>	经度偏移量	
<6>	经度半球	E（东经）或 W（西经）
<7>	高度偏移量	
<8>	坐标系代码	W84

注：以上协议中的*hh 表示验证码，*固定为验证码开始标识符，hh 为两位的验证码。<CR><LF>表示回车换行，即 0x0D0x0A。

29.2.3　GARMIN 定义的语句

NEMA0183 厂商扩展语句的典型代表是 GARMIN（美国高明）公司在 NEMA0183 标准下扩展的自定义 NEMA 语句。由于其在 GPS 行业中的地位，所以本小节将其定义的几条常用语句做个简单介绍，如表 29-2 所示。

表 29-2　GARMIN语句名称及格式

位　　置	数据项含义	数据项取值说明
估计误差信息（PGRME） $PGRME,<1>,M,<2>,M,<3>,M*hh<CR><LF>		
<1>	HPE（水平估计误差）	（0.0～999.9m）
<2>	VPE（垂直估计误差）	（0.0～999.9m）
<3>	EPE（位置估计误差）	（0.0～999.9m）
GPS 定位信息（PGRMF）$PGRMF,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>,<13>,<14>,<15>*hh<CR><LF>		
<1>	GPS 周数	（0～1023）
<2>	GPS 秒数	（0～604799）
<3>	UTC 日期	ddmmyy（日月年）格式
<4>	UTC 时间	hhmmss（时分秒）格式
<5>	GPS 跳秒数	整型
<6>	纬度	ddmm.mmmm（度分）格式，前面的 0 也将被传输
<7>	纬度半球	N（北半球）或 S（南半球）
<8>	经度	dddmm.mmmm（度分）格式，前面的 0 也将被传输
<9>	经度半球	E（东经）或 W（西经）
<10>	模式	M=手动，A=自动
<11>	定位类型	0=没有定位，1=2D 定位，2=3D 定位
<12>	地面速率	（0～1851 km/h）
<13>	地面航向	（000～359°），以正北为参考基准
<14>	PDOP 位置精度因子	（0～9），四舍五入取整
<15>	TDOP 时间精度因子	（0～9），四舍五入取整

续表

位　　置	数据项含义	数据项取值说明
坐标系统信息（PGRMM） $PGRMM,<1>*hh<CR><LF>		
<1>	当前使用的坐标系名称	数据长度可变，如 WGS 84，该信息在与 MapSource 进行实时连接的时候使用。MapSource 是 GARMI 公司提供的地图支持组件
工作状态信息（PGRMT） $PGRMT,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>*hh<CR><LF> 注：本语句每分钟发送一次，与所选择的波特率无关		
<1>	产品型号和软件版本	数据长度可变，如 GPS 15L/15H VER 2.05
<2>	ROM 校验测试	P=通过，F=失败
<3>	接收机不连续故障	P=通过，F=失败
<4>	存储的数据	R=保持，L=丢失
<5>	时钟的信息	R=保持，L=丢失
<6>	振荡器不连续漂移	P=通过，F=检测到过度漂移
<7>	数据不连续采集	C=正在采集，如果没有采集则为空
<8>	GPS 接收机温度	单位为摄氏度
<9>	GPS 接收机配置数据	R=保持，L=丢失
三维速度信息（PGRMV） $PGRMV,<1>,<2>,<3>*hh<CR><LF>		
<1>	东向速度	（514.4～514.4 m/s）
<2>	北向速度	（514.4～514.4 m/s）
<3>	上向速度	（999.9～9999.9 m/s）
信标差分信息（PGRMB） $PGRMB,<1>,<2>,<3>,<4>,<5>,K,<6>,<7>,<8>*hh<CR><LF>		
<1>	信标站频率	（0.0，283.5～325.0kHz，间隔为 0.5kHz）
<2>	信标比特率	（0，25，50，100 或 200bps）
<3>	SNR 信标信号信噪比	（0～31）
<4>	信标数据质量	（0～100）
<5>	与信标站的距离	单位为 km
<6>	信标接收机的通信状态	0=检查接线，1=无信号，2=正在调谐，3=正在接收，4=正在扫描
<7>	差分源	R=RTCM，W=WAAS，N=非差分定位
<8>	差分状态	A=自动，W=仅为 WAAS，R=仅为 RTCM，N=不接收差分信号

注：以上协议中的*hh 表示验证码，*固定为验证码开始标识符，hh 为两位的验证码。<CR><LF>表示回车换行，即 0x0A0x0D。

29.2.4　NEMA0183 协议的 TEXT 文本格式

NEMA0183 协议还提供了 TEXT 文本格式的协议，其数据主要由 5 部分组成，分别是协议头、时间、定位信息、速度信息和协议尾。具体的数据格式如表 29-3 所示。

表 29-3　NEMA0183 协议的TEXT文本格式

信 息 类 型	区 域 描 述	长度	注　　释
协议头	句头起始符	1	始终为@
时间	年	2	UTC 年的最后两位数字
	月	2	UTC 月, 01～12
	日	2	UTC 日, 01～31
	时	2	UTC 时, 00～23
	分	2	UTC 分, 00～59
	秒	2	UTC 秒, 00～59
定位信息	纬度半球	1	N（北纬）或 S（南纬）
	纬度坐标	7	WGS84 坐标系统，坐标格式为 ddmmmmm，在第四位数字后省略了一个小数点
	经度半球	1	E（东经）或 W（西经）
	经度坐标	8	WGS84 坐标系统，坐标格式为 dddmmmmm，在第五位数字后省略了一个小数点
	定位状态	1	d 表示二维差分定位；D 表示三维差分定位；g 表示二维定位；G 表示三维定位；S 表示模拟状态；_表示无效
	水平定位误差	3	单位为 m
	高度符号	1	+或–
定位信息	高度	5	海拔高，单位为 m
速度信息	东/西速度方向	1	E（东）或 W（西）
	东/西速度	4	单位是 m/s，在第三位后省略了一个小数点，如 1234 代表 123.4 m/s
	南/北速度方向	1	S（南） 或 N（北）
	南/北速度	4	单位是 m/s，在第三位后省略了一个小数点，如 1234 代表 123.4 m/s
	垂直速度方向	1	U（上）或 D（下）
	垂直速度	4	单位是 m/s，在第二位后省略了一个小数点，如 1234 代表 12.34 m/s
协议尾	句尾结束符	2	回车 0x0D 和换行 0x0A

下面的例子，表示接收到一条 GPS 信息，时间是 2008 年 8 月 28 日上午 8 点 38 分 52 秒，位置处在北纬 36°15.254′，东经 120°14.952′，定位信息是二维差分定位信息，水平定位误差为 13m，高度为+521m，方向向东 10.4m/s，向北 20.8m/s，向上是 1m/s。

```
@080828083852N3615254E12014952d013+00521E0104N2080U0001\n\r
```

29.3　串口接收 GPS 信息程序设计

前面讲述了有关 GPS 监控系统、GPS 定位知识以及 GPS 数据协议的知识，本节就结合前面讲解的知识，实际动手开发一个通过串口来接收 GPS 信息的程序。

29.3.1　实例背景

在本节的实例中，主要讲述如何通过串口读取 GPS 信息。实例主要分为两部分，一部分是 GPS 接收模块，另一部分是上端应用程序。GPS 接收模块使用 u-blox GPS Module。上端应用程序在 Windows 平台下使用 Visual Studio 2010 进行开发，其结构如图 29-2 所示。

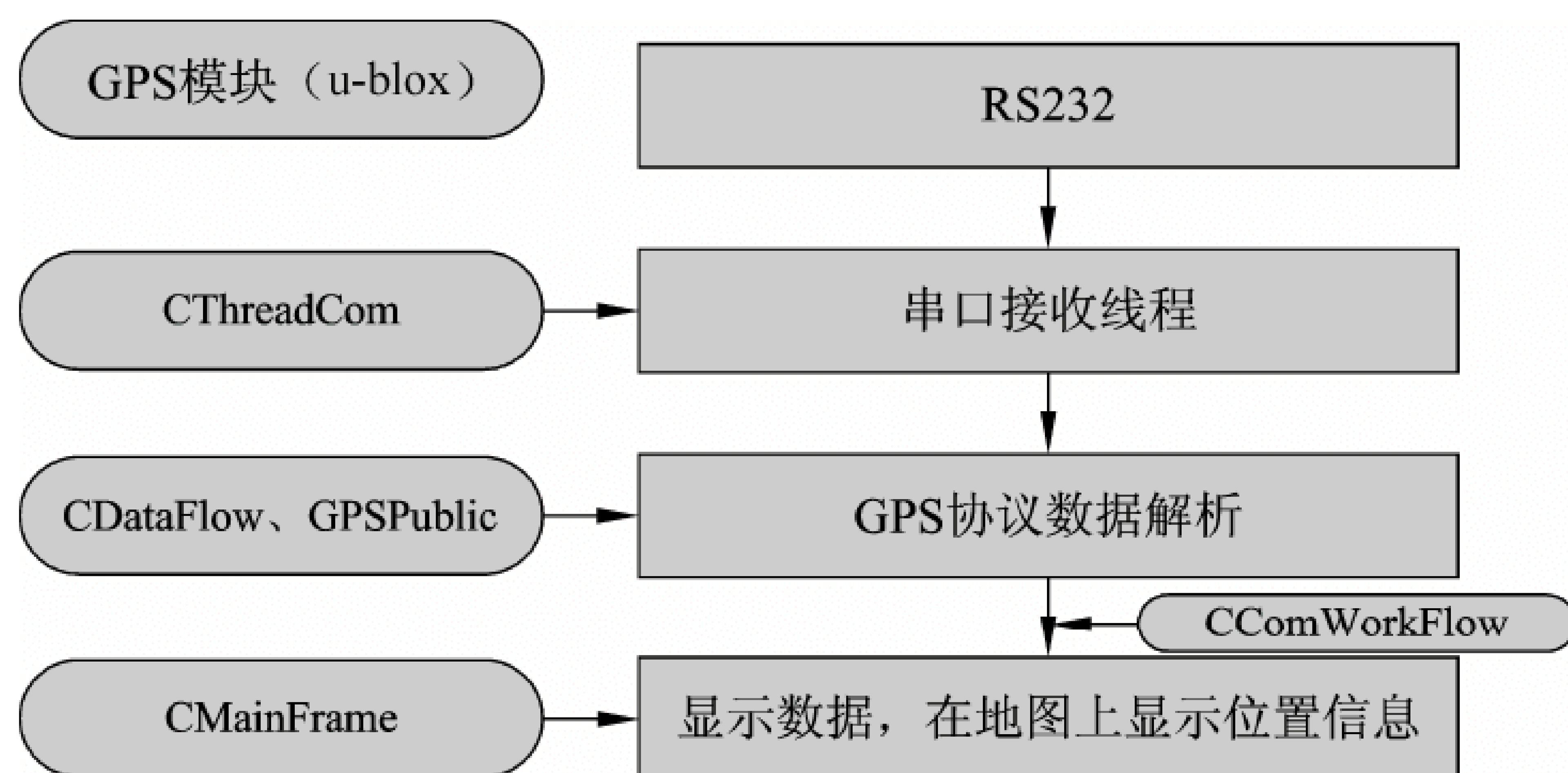


图 29-2　串口读取 GPS 数据的流程图

从图 29-2 可以看出，GPS 模块通过 RS232 模块与计算机的串口相连。上端应用程序开启串口接收线程，进行串口数据的读取。读取成功后，将其转给 CDataFlow 类进行协议解析。解析完成后，将原来的数据和解析后的数据发送给程序主窗体，由其在信息界面和地图界面上将位置信息显示出来。

29.3.2　GPS 模块与串口的通信协议

本实例分别对 NEMA0183 的文本格式、NEMA0183 的标准语句和 NEMA0183 的 GARMIN 语句进行解析处理。

29.3.3　程序功能

程序的主要功能如下。

- 协议数据的模拟发送：当硬件设备没有准备好时，可以使用此功能模拟发送 NEMA0183 的各条语句。
- 串口收发数据的查看：将串口发送的数据和接收的数据显示在界面上。
- 解析后的 NEMA0183 语句查看：在界面上显示所有解析后的 NEMA0183 语句。
- 位置信息查看：在界面上显示所有解析后的位置信息。
- 地图上显示 GPS 位置信息：在地图上显示 GPS 位置信息。

29.3.4　界面设计

本实例在界面设计上主要分为 5 部分，如图 29-3 所示。

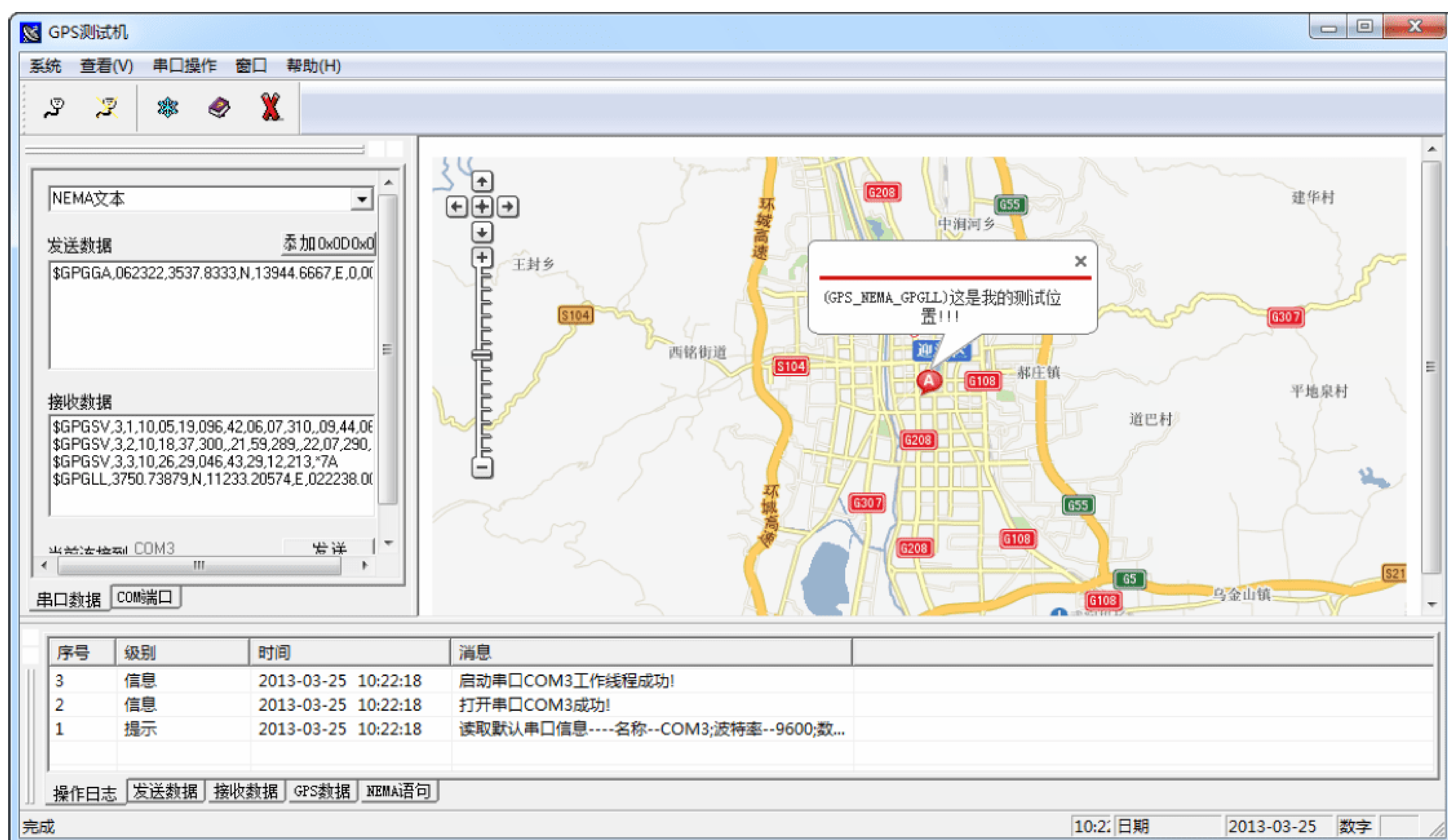


图 29-3　程序主界面

- ❑ 菜单：其上放置常用的菜单项，如打开串口、关闭串口、清除日志等。
- ❑ 工具栏：其上放置与菜单项对应的常用的工具按钮，其功能与菜单条上的菜单项相对应。
- ❑ 串口工作区：如图 29-3 左边的对话框，其中有两页，一页是串口数据对话框，显示当前工作串口的发送数据和接收数据的信息，如图 29-4 所示；另一页是 COM 端口对话框，其中显示当前工作的所有串口的配置信息，如图 29-5 所示。



图 29-4　串口数据对话框

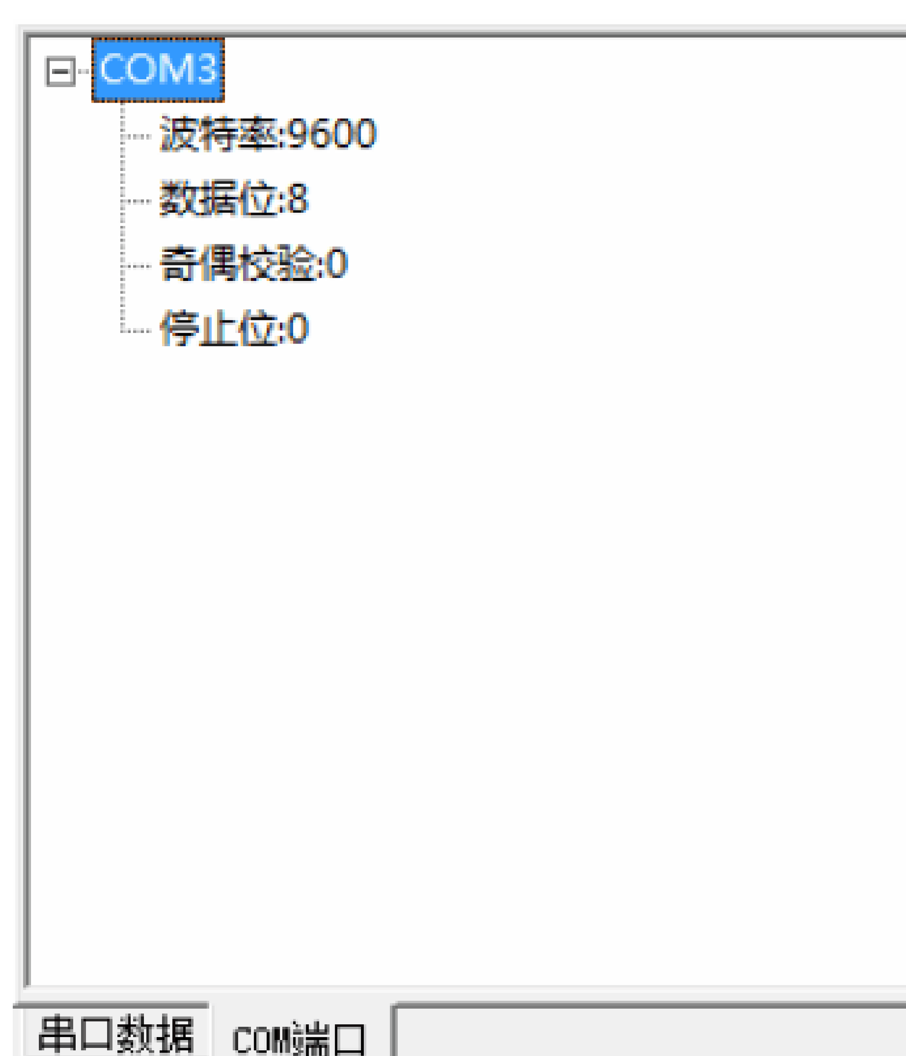


图 29-5　COM 端口对话框

- ❑ 地图对话框：主要用于在地图上显示接收到的 GPS 位置信息。
- ❑ 信息对话框：主要用于显示串口的工作信息。其上主要有 5 页。第一页是日志信息，显示操作日志，如图 29-6 所示；第二页是发送数据日志页，显示串口发送的数据信息，如图 29-7 所示；第三页是接收数据日志页，显示串口接收的数据信息，

如图 29-8 所示；第四页是 GPS 数据页，显示接收到的 GPS 信息的数据，如图 29-9 所示；第五页是 NEMA 语句页，显示接收到的 NEMA 语句信息，如图 29-10 所示。

序号	级别	时间	消息
3	信息	2013-03-25 10:22:18	启动串口COM3工作线程成功!
2	信息	2013-03-25 10:22:18	打开串口COM3成功!
1	提示	2013-03-25 10:22:18	读取默认串口信息----名称--COM3;波特率--9600;数...

操作日志 | 发送数据 | 接收数据 | GPS数据 | NEMA语句

图 29-6　操作日志页

序号	时间	串口	发送内容/发送结果
4	COM3	2013-03-25 10:25:20	No Error
3	COM3	2013-03-25 10:25:19	$GPRMC,121252.000,A,3958.3032,N,11629.6046,...
2	COM3	2013-03-25 10:25:13	No Error
1	COM3	2013-03-25 10:25:13	$GPGGA,062322,3537.8333,N,13944.6667,E,0,00,...

操作日志 | 发送数据 | 接收数据 | GPS数据 | NEMA语句

图 29-7　发送数据日志页

序号	串口	时间	接收到的数据
66	COM3	2013-03-25 10:26:58	$GPGSV,3,3,10,26,27,047,41,29,10,211,*77$GPGL...
65	COM3	2013-03-25 10:26:58	$GPGSV,3,1,10,05,18,097,42,06,07,309,21,09,43,0...
64	COM3	2013-03-25 10:26:58	$GPGGA,022700.00,3750.74143,N,11233.20885,E...
63	COM3	2013-03-25 10:26:58	$GPVTG,,T,,M,0.166,N,0.307,K,A*26
62	COM3	2013-03-25 10:26:58	$GPRMC,022700.00,A,3750.74143,N,11233.2088...
61	COM3	2013-03-25 10:26:57	$GPGLL,3750.74126,N,11233.20881,E,022659.00

操作日志 | 发送数据 | 接收数据 | GPS数据 | NEMA语句

图 29-8　接收数据日志页

编号	串口	GPS时间	经度	纬度	海拔	定位状态	水平精度	X方向	Y速度	Z速度	其他信息
42	COM3	02:27:28	东经112.5535...	北纬37.8456...		有效定位					模式=自主定...
41	COM3	02:27:28	东经112.5535...	北纬37.8456...	815.200000...	非差分定位	6.300000				卫星数目4--...
40	COM3	2013-03-25 02:27:28	东经112.5535...	北纬37.8456...		有效定位		方向=78.17...			磁偏角=0.00...
39	COM3	02:27:27	东经112.5535...	北纬37.8456...		有效定位					模式=自主定...
38	COM3	2013-03-25 02:27:27	东经112.5535...	北纬37.8456...		有效定位		方向=0.000...			磁偏角=0.00...
37	COM3	02:27:26	东经112.5535	北纬37.8456		有效定位					模式=自主定

操作日志 | 发送数据 | 接收数据 | GPS数据 | NEMA语句

图 29-9　GPS 数据页

序号	串口来源	NEMA语句内容
10	COM3	总数=3----编号=3----可见=10--------PRN码=26,卫星仰角=27,卫星方位角=47,信噪比=39----PRN码=29,卫...
9	COM3	模式=自动----定位类型=3D定位----PRN=卫星26,卫星5,卫星9,卫星15,----PDOP=8.860000----HDOP=6.33...
8	COM3	总数=3----编号=3----可见=10--------PRN码=26,卫星仰角=27,卫星方位角=47,信噪比=39----PRN码=29,卫...
7	COM3	总数=3----编号=2----可见=10--------PRN码=18,卫星仰角=39,卫星方位角=301,信噪比=23----PRN码=21,...
6	COM3	真北航向=0----磁北航向=0----地面速率(节)=0.166000----地面速率(公里/小时)=0.307000----模式指示=自...
5	COM3	总数=3----编号=3----可见=10--------PRN码=26,卫星仰角=27,卫星方位角=47,信噪比=39----PRN码=29,卫

操作日志 | 发送数据 | 接收数据 | GPS数据 | NEMA语句

图 29-10　NEMA 语句页

29.3.5　结构声明

在进行协议解析时，用到的核心结构就是存储协议数据的共用体。共用体 GPSPack 中

存储了与协议解析有关的结构变量。其基本形式如下：

```
01  union GPSPack
02  {
03      struct 各种类型的 NEMA 语句
04      {
05          char head;        //语句头
06          char bodyType;    //各种类型的 NEMA 语句的语句体 body
07          //语句体，根据语句类型不同，存储不同的结构体
08          char checkBegin;
09          //校验码标识，如果是文本格式语句，则没有此项
10          char check[2];
11          //语句校验码，如果是文本格式语句，则没有此项
12          char tail[2];     //语句尾
13      }GPS_各种类型的 NEMA 语句;
14      ...
15  }
```

在共用体中的每种结构中存储了不同类型的 NEMA0183 语句的信息。下面以文本语句和 NEMA0183 标准语句为例，说明其结构体的定义。对于厂商扩展语句，用户可以根据自己设备对应的数据协议格式，定义相应的结构体。在本实例附带的源代码中，附有美国高明公司的厂商自定义语句的结构定义。

结构体 tagBODY_NEMA_TEXT 保存 NEMA 文本格式的协议数据项，包括日期、时间、定位的经纬度、定位误差、海拔、三维速度和方向数据项。声明代码如下：

```
01  struct tagBODY_NEMA_TEXT                    //NEMA 的文本方式的数据体
02  {
03      tagUTC_DATE      date;                  //UTC 日期
04      tagUTC_TIME      time;                  //UTC 时间
05      char             latitudeType;          //纬度半球
06      tagLatitude      latitude;              //纬度坐标
07      char             longitudeType;         //经度半球
08      tagLongitude     longitude;             //经度坐标
09      char             gpsStatus;             //定位状态
10      double           hdop;                  //水平定位误差
11      char             altitudeSymbol;        //高度符号
12      tagAlitude       altitude;              //高度
13      char             xDirect;               //东/西速度方向
14      double           xVolity;               //东/西速度
15      char             yDirect;               //南/北速度方向
16      double           yVolity;               //南/北速度
17      char             zDirect;               //垂直速度方向
18      double           zVolity;               //垂直速度
19  };
```

结构体 tagBODY_NEMA_GPGGA 保存 GPS 定位信息语句（$GPGGA）的协议数据项，包括日期、经纬度、GPS 状态、卫星数目、水平精度、海拔和差分信息等数据项。声明代码如下：

```
01  struct tagBODY_NEMA_GPGGA
02  {
03      tagUTC_TIME          time;
04      //<1> UTC 时间，hhmmss（时分秒）格式
```

```
05      tagLatitude          latitude;
06      //<2> 纬度 ddmm.mmmm（度分）格式（前面的 0 也将被传输）
07      char               latitudeType;
08      //<3> 纬度半球 N（北半球）或 S（南半球）
09      tagLongitude       longitude;
10      //<4> 经度 dddmm.mmmm（度分）格式（前面的 0 也将被传输）
11      char               longitudeType;
12      //<5> 经度半球 E（东经）或 W（西经）
13      char               gpsStatus;
14      //<6> GPS 状态：0=未定位，1=非差分定位，2=差分定位，6=正在估算
15      int                  sateCount;
16      //<7> 正在使用解算位置的卫星数量（00~12）（前面的 0 也将被传输）
17      double               hdop;
18      //<8> HDOP 水平精度因子（0.5~99.9）
19      tagAlitude           altitude;
20      //<9> 海拔高度（-9999.9~99999.9）
21      double               height;
22      //<10> 地球椭球面相对大地水准面的高度
23      int                  diffSecond;
24      //<11>差分时间（从最近一次接收到差分信号开始的秒数，如果不是差分定位则为空）
25      char               diffStationID[4];
26      //<12>差分站 ID 号 0000~1023（前面的 0 也将被传输，如果不是差分定位则为空）
27  };
```

结构体 tagBODY_NEMA_GPGSA 保存当前卫星信息语句（$GPGSA）的协议数据项，包括 GPS 定位模式、定位类型、三维精度因子和 PRN 码等数据项。声明代码如下：

```
01  struct tagBODY_NEMA_GPGSA
02  {
03      char               gpsmode;
04      //<1> 模式，M=手动，A=自动
05      char               postype;
06      //<2> 定位类型，1=没有定位，2=2D 定位，3=3D 定位
07      int                  prn[MAX_SATE_TOTAL];
08      //<3> PRN 码（伪随机噪声码），
09      //正在用于解算位置的卫星号（01~32，前面的 0 也将被传输）
10      double               pdop;
11      //<4> PDOP 位置精度因子（0.5~99.9）
12      double               hdop;
13      //<5> HDOP 水平精度因子（0.5~99.9）
14      double               vdop;
15      //<6> VDOP 垂直精度因子（0.5~99.9）
16  };
```

结构体 tagBODY_NEMA_GPGSV 保存可见卫星信息语句（$GPGSV）的协议数据项，包括语句总数、语句编号、可见卫星数、PRN 码及其仰角、方位角和信噪比等数据项。声明代码如下：

```
01  struct tagBODY_NEMA_GPGSV
02  {
03      int                  total;
04      //<1> GSV 语句的总数
05      int                  no;
06      //<2> 本句 GSV 的编号
07      int                  validsate;
08      //<3> 可见卫星的总数（00~12，前面的 0 也将被传输）
```

```
09      int                     prn[MAX SATE ONE TOTAL];
10      //<4> PRN 码（伪随机噪声码）（01~32，前面的 0 也将被传输）
11      int                     yj[MAX SATE ONE TOTAL];
12      //<5> 卫星仰角（00~90°，前面的 0 也将被传输）
13      int                     fwj[MAX_SATE_ONE_TOTAL];
14      //<6> 卫星方位角（000~359°，前面的 0 也将被传输）
15      int                     xzb[MAX SATE ONE TOTAL];
16      //<7> 信噪比（00~99dB，没有跟踪到卫星时为空，前面的 0 也将被传输）
17  };
```

结构体 tagBODY_NEMA_GPRMC 保存推荐定位信息语句（$GPRMC）的协议数据项，包括日期、时间、定位状态、经纬度、速度、方向、磁偏角、磁偏角方向和模式等数据项。声明代码如下：

```
01  struct tagBODY NEMA GPRMC
02  {
03      tagUTC TIME             time;
04      //<1> UTC 时间，hhmmss（时分秒）格式
05      char                posvalid;
06      //<2> 定位状态，A=有效定位，V=无效定位
07      tagLatitude             latitude;
08      //<3> 纬度 ddmm.mmmm（度分）格式（前面的 0 也将被传输）
09      char                latitudeType;
10      //<4> 纬度半球 N（北半球）或 S（南半球）
11      tagLongitude        longitude;
12      //<5> 经度 dddmm.mmmm（度分）格式（前面的 0 也将被传输）
13      char                longitudeType;
14      //<6> 经度半球 E（东经）或 W（西经）
15      double                  volity;
16      //<7> 地面速率（000.0~999.9 节，前面的 0 也将被传输）
17      double                  direct;
18      //<8> 地面航向（000.0~359.9°，以真北为参考基准，前面的 0 也将被传输）
19      tagUTC DATE             date;
20      //<9> UTC 日期，ddmmyy（日月年）格式
21      double                  cpj;
22      //<10> 磁偏角（000.0~180.0°，前面的 0 也将被传输）
23      char                cpjdirect;
24      //<11> 磁偏角方向，E（东）或 W（西）
25      char                mode;
26      //<12> 模式指示（仅 NEMA0183 3.00 版本输出
27      //A=自主定位，D=差分，E=估算，N=数据无效）
28  };
```

结构体 tagBODY_NEMA_GPVTG 保存地面速度信息语句（$GPVTG）的协议数据项，包括方向、速度和模式等数据项。声明代码如下：

```
01  struct tagBODY NEMA GPVTG
02  {
03      int                     directtn;
04      //<1> 以真北为参考基准的地面航向（000~359°，前面的 0 也将被传输）
05      int                     directcn;
06      //<2> 以磁北为参考基准的地面航向（000~359°，前面的 0 也将被传输）
07      double                  volityj;
08      //<3> 地面速率（000.0~999.9 节，前面的 0 也将被传输）
09      double                  volitykm;
10      //<4> 地面速率（0000.0~1851.8 公里/小时，前面的 0 也将被传输）
11      char                mode;
```

```
12      //<5> 模式指示（仅 NEMA0183 3.00 版本输出
13      //A=自主定位，D=差分，E=估算，N=数据无效）
14  };
```

结构体 tagBODY_NEMA_GPGLL 保存定位地理信息语句（$GPGLL）的协议数据项，包括经纬度、时间、定位状态和模式等数据项。声明代码如下所示：

```
01  struct tagBODY_NEMA_GPGLL
02  {
03       tagLatitude          latitude;
04      //<1> 纬度 ddmm.mmmm（度分）格式（前面的 0 也将被传输）
05       char                 latitudeType;
06      //<2> 纬度半球 N（北半球）或 S（南半球）
07       tagLongitude         longitude;
08      //<3> 经度 dddmm.mmmm（度分）格式（前面的 0 也将被传输）
09       char                 longitudeType;
10      //<4> 经度半球 E（东经）或 W（西经）
11       tagUTC_TIME          time;
12      //<5> UTC 时间，hhmmss（时分秒）格式
13       char                 posvalid;
14      //<6> 定位状态，A=有效定位，V=无效定位
15       char                 mode;
16      //<7> 模式指示（仅 NEMA0183 3.00 版本输出
17      //A=自主定位，D=差分，E=估算，N=数据无效）
18  };
```

结构体 tagBODY_NEMA_GPZDA 保存时间和日期信息语句（$GPZDA）的协议数据项，包括日期和时间数据项。声明代码如下：

```
01  struct tagBODY_NEMA_GPZDA
02  {
03      tagUTC_TIME          time;
04      //<1> UTC 时间，hhmmss（时分秒）格式
05      tagUTC_DATE          date;
06      //<2> UTC 日期，日   <3> UTC 日期，月   <4> UTC 日期，年
07  };
```

结构体 tagBODY_NEMA_GPDTM 保存大地坐标系信息语句（$GPDTM）的协议数据项，包括坐标系、经纬度偏移量和高度偏移量等数据项。声明代码如下：

```
01  struct tagBODY_NEMA_GPDTM         //大地坐标系信息
02  {
03      char*    localdatum;          //<1>本地坐标系代码 W84
04      char*    subdatum;            //<2>坐标系子代码 空
05      double   latitudeoffset;      //<3>纬度偏移量
06      char latitudetype;            //<4>纬度半球 N（北半球）或 S（南半球）
07      double   longitudeoffset;     //<5>经度偏移量
08      char longitudetype;           //<6>经度半球 E（东经）或 W（西经）
09      double   alitudeoffset;       //<7>高度偏移量
10      char*    datum;               //<8>坐标系代码 W84
11  };
```

29.3.6　初始化操作

在本实例中，每个串口开启一个工作线程 CThreadCom 类，主要完成串口的字符接收

监测，同时可以打开串口、关闭串口以及发送数据等。

在进行 GPS 信息接收前，首先需要进行初始化操作，即打开并配置工作串口。在 Win32 下，串口被当作文件来处理，因此打开串口使用 CreateFile()函数来实现。打开串口后，需要设置串口的工作缓冲区的大小，包括输入缓冲区大小、输出缓冲区大小；配置串口工作参数，如波特率、数据位、停止位和奇偶校验等；配置串口工作超时时间；配置线程检测的串口事件。

在本实例中，使用 OpenComm()函数来初始化串口。该函数的参数分别为要打开的串口名称、串口的工作参数、串口所属的数据处理对话框、串口线程与数据串口之间的通信消息、串口所属的父窗体以及串口与父窗体之间的通信消息。该函数的返回值为 BOOL 型，表示打开串口是否成功。具体代码如下：

```
01  BOOL CThreadCom::OpenCom( CString com, DCB dcb,
02                            CWnd *pWndData,DWORD dwMsgToWndData,
03                            CWnd *pWndParent, DWORD dwMsgToParent)
04  {
05      //将传递进来的变量，赋值给成员变量
06      m_Com = com;
07      m hCom=INVALID HANDLE VALUE;
08      m sError="No Error";
09      m pWndData       =   pWndData;
10      m_dwMsgToData    =   dwMsgToWndData;
11      m pWndParent=    pWndParent;
12      m dwMsgToParent =    dwMsgToParent;
13      m hCom=::CreateFile(m Com,  GENERIC READ
14          |GENERIC_WRITE,0,NULL,OPEN_EXISTING,
15          FILE FLAG OVERLAPPED,NULL);                        //打开串口
16      if (m_hCom==INVALID_HANDLE_VALUE)                      //判断处理结果
17      {
18          m_sError.Format("Open %s Error", m_Com);
19          return false;
20      }
21      //设置串口的输入缓冲区大小和输出缓冲区大小，建议在设置时，将其定义为宏
22      ::SetupComm(m_hCom, MAX_COM_IN_LENGTH, MAX_COM_OUT_LENGTH);
23      //设置串口的工作参数，在 NEMA0183 中规定，
24      //波特率为 4800 及其以上，数据位为 8 位，无奇偶校验，停止位为 1 位
25      DCB tempdcb;
26      int nSuccess = GetCommState(m_hCom,&tempdcb);//获取串口参数配置
27      if (!nSuccess)                                     //判断操作结果
28      {
29          m_sError="Can't get commstate!";               //保存错误信息
30          CloseHandle(m_hCom);                           //关闭串口
31          m_hCom=INVALID_HANDLE_VALUE;                   //赋值串口句柄
32          return false;                                  //返回
33      }
34      tempdcb.BaudRate=   dcb.BaudRate;                  //设置串口波特率
35      tempdcb.ByteSize=   dcb.ByteSize;                  //设置字节大小
```

```
36      tempdcb.Parity      =   dcb.Parity;          //设置奇偶校验位
37      tempdcb.StopBits=   dcb.StopBits;            //设置停止位
38      if (SetCommState(m_hCom,&tempdcb)==0)        //设置串口参数
39      {                                            //判断操作结果
40         CloseHandle(m_hCom);                      //保存错误信息
41         m_hCom=INVALID_HANDLE_VALUE;              //关闭串口
42         m_sError.Format("Init %s Error!", m_Com);//赋值串口句柄
43         return false;                             //返回
44      }
45      //设置串口的工作超时参数，此处设置其读超时参数和写超时参数
46      ::GetCommTimeouts(m_hCom,&m_Commtimeout);
47      m_Commtimeout.ReadTotalTimeoutMultiplier=5;
48      m_Commtimeout.ReadTotalTimeoutConstant=100;
49      m_Commtimeout.WriteTotalTimeoutMultiplier=0;
50      m_Commtimeout.WriteTotalTimeoutConstant=1024;
51      ::SetCommTimeouts(m_hCom, &m_Commtimeout);
52      //设置串口检测的事件
53      DWORD CommMask;
54      CommMask=0
55         |EV_BREAK    //数据输入时，检测到中断
56         |EV_CTS      //CTS 信号改变状态
57         |EV_DSR      //DSR 信号改变状态
58         |EV_ERR//发生行状态错误，包括 CE_FRAME、CE_OVERRUN 和 CE_RXPARITY
59         |EV_EVENT1
60         |EV_EVENT2
61         |EV_PERR     //发生打印错误 A printer error occured
62         |EV_RING     //检测到振铃
63         |EV_RLSD     //RLSD 信号改变状态
64         |EV_RX80FULL //接收缓冲区中已经占用 80%以上
65         |EV_RXCHAR   //接收到字符，并将其放入输入缓冲区中
66         |EV_RXFLAG   //接收到事件字符，并将其放入到输入缓冲区中
67         |EV_TXEMPTY; //输出缓冲区中最后一个字符发送出去
68      if (!SetCommMask(m_hCom, CommMask))
69         return false;
70      m_bOpen=true;
71      return true;    //一切成功，返回
72   }
```

29.3.7　GPS 数据接收的实现方法

在打开串口后，就可以进行 GPS 数据的接收。在 Win32 下，读取串口数据使用 ReadFile()函数来实现。对串口的读取，需要注意异步读取的控制。当串口在异步读取未完成时，需要处理并等待数据的读取。

在本实例中，数据接收部分由 Run()线程运行函数和 ReceiveData()数据接收函数实现，其工作流程如图 29-11 所示。

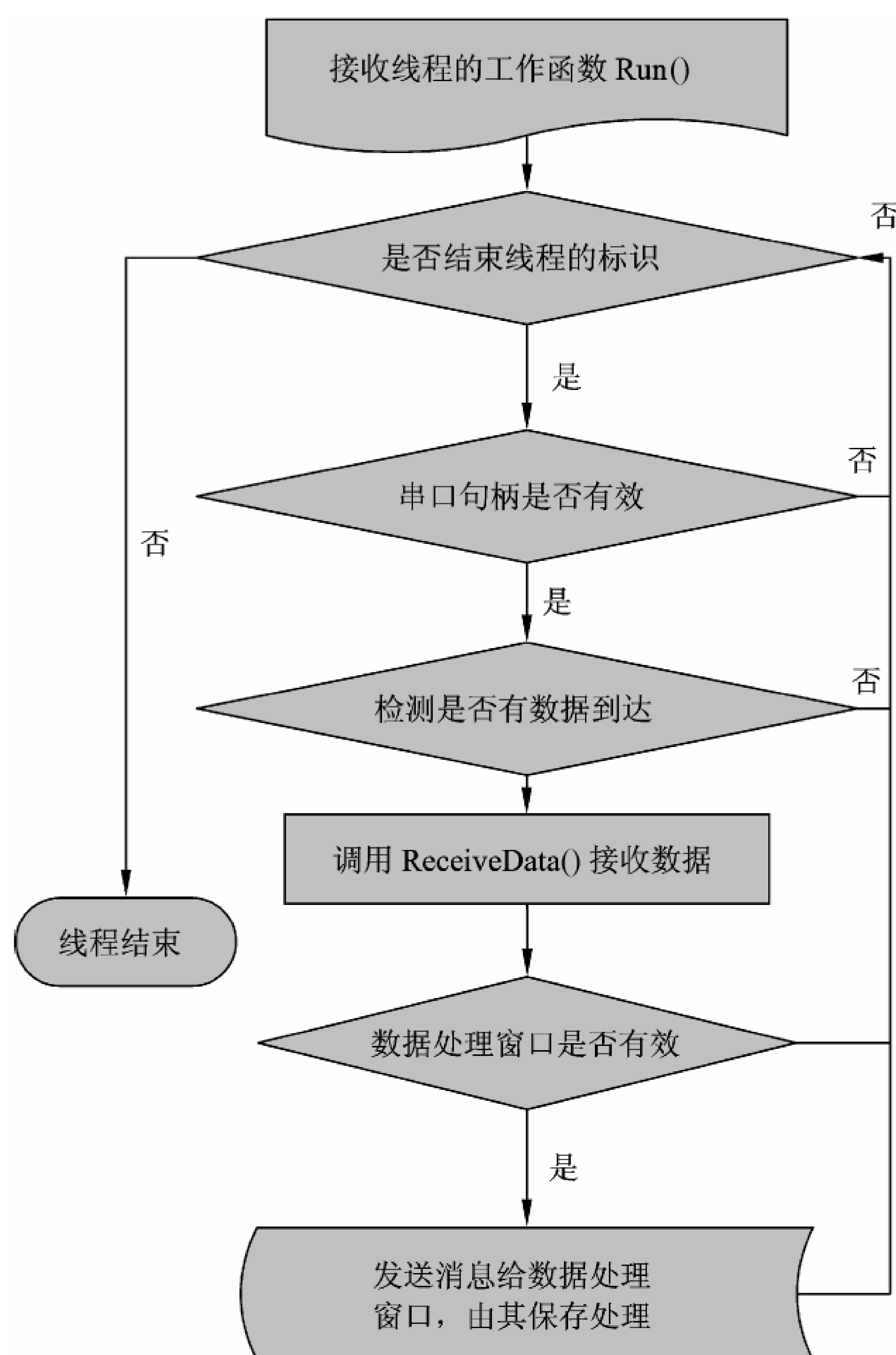


图 29-11　串口数据接收函数流程图

1. 接收线程的Run()工作函数

该函数实现对应的串口的接收事件的监测，当监测到有数据到达时，执行 ReceiveData()函数接收数据，并将接收到的函数消息通知给数据处理窗体。该函数没有传入参数。该函数返回值为线程结束时的结束代码。其代码如下：

```
01  int CThreadCom::Run()                              //线程运行函数
02  {
03      DWORD       dwEvent;                           //串口事件
04      int         nLength ;                          //接收长度临时变量
05      BYTE     rBuf[MAX_COM_IN_LENGTH+1];            //接收数据的缓冲区
06      while (!m_bDone)//如果线程工作结束标识值为 false，则检测串口数据，并接收
07      {
08          while (m_hCom!=INVALID_HANDLE_VALUE)
09          //如果串口句柄是有效值，则执行处理
10          {
11              dwEvent = 0 ;                          //检测串口事件
12              WaitCommEvent(m_hCom, &dwEvent, NULL);
13              //如果有字符接收串口事件，则接收串口数据
14              if ((dwEvent & EV_RXCHAR) == EV_RXCHAR)
```

```
15              {
16                  do                          //循环接收串口数据
17                  {
18                      if (nLength = ReceiveData((LPSTR)rBuf,
19                          MAX COM IN LENGTH) )
20                      {                       //接收串口数据
21                          //如果串口数据中有字符接收事件
22                          //则将事件传送给 CDataFlow 进行处理
23                          if(dwEvent & ~EV RXCHAR)
24                          {
25                              if(m pWndData)
26                              {
27                                  m pWndData->SendMessage(
28                                      m dwMsgToData,dwEvent,0);
29                              }
30                          }
31                          if(nLength)//接收到数据后传送给 CDataFlow 进行处理
32                          {
33                              if(m_pWndData)
34                              {
35                                  m pWndData->SendMessage(
36                                      m dwMsgToData,(DWORD)rBuf,nLength);
37                              }
38                          }
39                      }
40                  }while ( nLength > 0 );
41              }
42              Sleep(1);
43          }
44          Sleep(1);
45      }
46      CloseHandle( m overRead.hEvent);    //关闭读异步事件变量
47      return CWinThread::Run();           //调用线程运行函数
48  }
```

2. 串口数据ReceiveData()接收函数

该函数实现串口数据的接收。该函数传入参数为接收数据的缓冲区和接收的最大长度，函数会取这个值与当前缓冲区中字符数较小的数目来读取。该函数返回值表示接收串口数据是否正确，true 表示正确接收串口数据，false 表示接收串口数据失败。其代码如下：

```
01  BOOL CThreadCom::ReceiveData(LPCTSTR rBuf,
02      int nMaxLength) //接收数据
03  {
04      DWORD       dwLength;                       //接收的数据的长度
05      DWORD       dwError;                        //错误代码
06      DWORD       dwErrorFlags;                   //错误代码
07      COMSTAT  ComStat;                           //串口状态值
08      BOOL      fReadStat ;                       //从串口读数据的状态
09      //清除端口的错误信息
10      ClearCommError( m hCom, &dwErrorFlags, &ComStat);
11      //取接收字符串中的字符数目和空间大小的最小值作为要读取的字符数
12      dwLength = min( (DWORD) nMaxLength, ComStat.cbInQue ) ;
13      if (dwLength <= 0)
14          return dwLength;     //如果串口没有数据需要接收，则返回
15      //读取串口的数据
```

```
16      fReadStat = ReadFile( m_hCom, (void*)rBuf,
17          dwLength, &dwLength,&m overRead ) ;
18      if(fReadStat)
19          return dwLength;     //如果读串口操作成功，则返回
20      dwError = GetLastError();//如果读串口操作未成功，则获取其错误代码
21      if (dwError == ERROR IO PENDING)
22          //如果错误是串口读写异步未执行完，则继续读取
23      {
24          //首先应该判断接收是否完成
25          while(!GetOverlappedResult( m_hCom,
26              &m overRead, &dwLength, true ))
27          {
28              dwError = GetLastError();
29              if(dwError == ERROR IO INCOMPLETE)
30              {
31                  continue;     //如果发送还没有完成，则继续等待发送结束
32              }
33              else
34              { //如果发生错误，则处理错误
35                  m sError.Format("<%u>", dwError) ;
36                  ClearCommError(m hCom, &dwErrorFlags, &ComStat);
37                  if (dwErrorFlags > 0)
38                  {
39                      printf(m sError, "<%u>", dwErrorFlags);
40                  }
41                  break;
42              }
43          }
44      }
45      else //如果在读取的过程中发生其他错误，则处理错误信息
46      {
47          dwLength = 0;
48          m_sError.Format("<%u>", dwError) ;//如果发生其他错误，则处理错误
49          ClearCommError( m hCom, &dwErrorFlags, &ComStat ) ;
50          if (dwErrorFlags > 0)
51          {
52              printf(m sError, "%s<%u>", m sError, dwErrorFlags);
53          }
54      }
55      return dwLength;                     //返回接收到的数据长度
56  }
```

29.3.8　GPS 数据解析的实现方法

在接收到数据后，需要对数据按照协议格式进行解析。从上面的代码可以看出，串口数据接收线程在成功接收数据后，会将其发送给对应的数据处理窗体 CDataFlow 对象。CDataFlow 对象在接收到发送过来的消息后，会将数据存储到变量中，并对缓冲区中的数据进行解析，其工作流程如图 29-12 所示。

从流程图中可以看出，解析对象在接收到串口接收线程发送过来的接收到数据的消息后，会触发 OnComMsg()函数，此函数会存储数据，并调用 SearchFlow()函数来查找数据包。如果查找到有效数据包，则会调用 DealFlow()函数来处理数据包，并发送解码消息，由 OnDecodeMsg()解码处理函数来具体解码。下面分别详细说明各个函数的实现，由于篇

幅原因，有关具体的协议数据解析，仅以 GGA 数据包的解析过程函数 DecodeNEMA_GPGGA()为例讲述，其他协议语句的具体解析过程，请参见源代码。

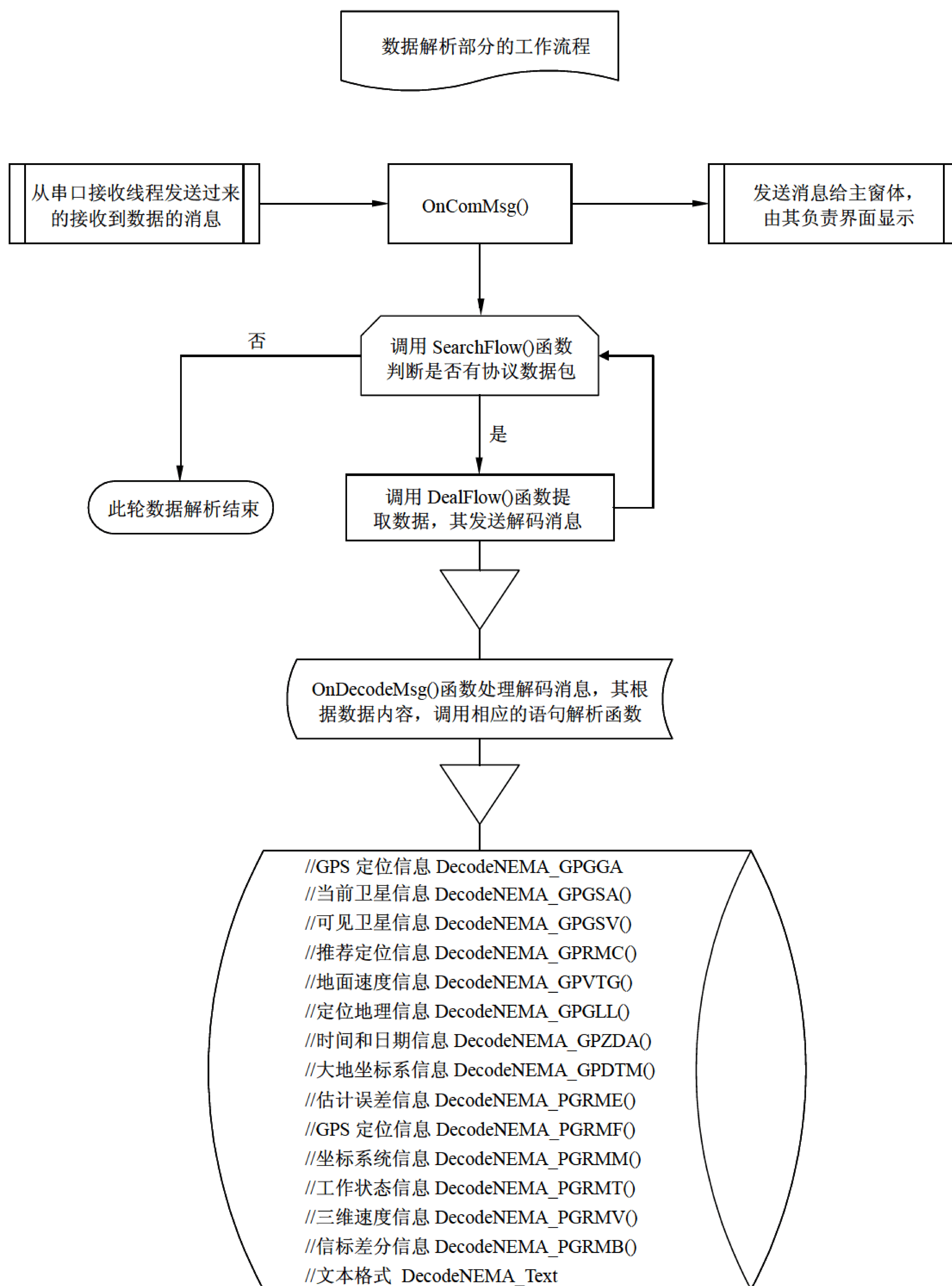


图 29-12　数据解析处理流程

1．接收数据消息处理的OnComMsg()函数

该函数处理串口接收到的数据。当串口数据接收线程监测到有数据到达时，会接收并发送消息给数据处理对话框。数据处理对话框就会调用此函数对其进行处理，主要是将数据存入待解析字符串中，发送消息给主窗体，由主窗体实现数据的界面显示，并调用解析函数进行协议数据的解析。该函数的传入参数是存储接收数据的数据缓冲区的指针和接收到的数据的长度。该函数无返回值。其代码如下：

```
01  //串口消息处理函数
02  void CDataFlow::OnComMsg(DWORD dwEvent, DWORD dwLen)
03  {
04      if(!dwLen)              //如果数据长度等于 0，则返回
05      {
06          m_dwComEvent=dwEvent;
07          return;
08      }
09      while(dwLen > 0)
10      {                       //将数据复制到本地数组中
11          BYTE s[MAX_COM_IN_LENGTH+1];
12          int len = min(MAX_COM_IN_LENGTH, dwLen);
13          memset(s, 0, sizeof(s));
14          memcpy(s,(BYTE * )dwEvent, len);
15          s[len]='\0';
16          dwLen -= len;
17          CString content = s;
18          //发送给主窗体，有数据到达，并显示
19          ::SendMessage(AfxGetApp()->GetMainWnd()->m_hWnd,
20              WM_RECEIVE_COM_DATA,
21              (WPARAM)&m_Com,  (LPARAM)&content);
22          for(DWORD i=0; i<len; i++)
23          {
24              m_Data.Add(s[i]);
25          }
26      }
27      while(SearchFlow())
28          DealFlow();
29      //判断是否查找到协议数据，查找到后处理协议数据
30      return;                 //函数返回
31  }
```

2．查找协议数据包的SearchFlow()函数

该函数实现从待解析数据缓冲区中查找协议数据包的功能。查找数据包的标准是判断协议头和协议尾是否与协议定义的相符，如果查找到，则将协议头的位置、协议尾的位置以及协议数据包的长度存入变量中。该函数没有传入参数。该函数返回值为 BOOL 型，表示是否查找到协议数据包。如果返回 true，表示数据缓冲区中存在协议数据包；如果返回 false，则表示当前的数据缓冲区中没有协议数据包。其代码如下：

```
01  BOOL CDataFlow::SearchFlow()    //查找协议数据
02  {
03      for (int i = 0;i < m_Data.GetUpperBound(); i++)
04      //循环数据缓冲区中的数据
05      {
06          if ((m_Data[i] == NEMA_HEAD_STANDAD)
```

```
07              || (m Data[i] == NEMA HEAD TEXT))
08          {                          //检测到数据头后，将其位置记录下来
09              m dwDataGramHead = i;
10          }
11          if ((m_Data[i] == NEMA_TAIL_STANDAD_1)
12              && (m_Data[i+1] == NEMA_TAIL_STANDAD_2) )
13          {
14              //检测到数据尾后，将其位置记录下来
15              m dwDataGramTail = i + 1;
16              m dwDataGramLen =
17                  m dwDataGramTail-m dwDataGramHead+1;
18              return true;           //检测到协议数据，返回 true
19          }
20      }
21      return false;                  //未检测到协议数据，返回 false
22  }
```

3．协议数据包解析的DealFlow()函数

该函数实现从待解析数据缓冲区中提取协议数据包的功能。当在数据缓冲区中查找到协议数据包后，调用此方法将协议数据包从缓冲区中提取出来，并发送消息给解码函数进行解码，该函数没有传入参数也没有返回值。其代码如下：

```
01  void CDataFlow::DealFlow()                            //处理协议数据
02  {
03      //判断数据有效性
04      if ((m dwDataGramLen < 0) || (m dwDataGramLen >
05          MAX NEMA SENTENCE MAX LENGTH))
06      {
07          return;
08      }
09      BYTE gramBytes[MAX_NEMA_SENTENCE_MAX_LENGTH+1];//定义报文字节数组
10      memset(gramBytes, 0x00, sizeof(gramBytes));     //初始化字节数组
11      for (int i = 0;i < m dwDataGramLen;i++)         //依次获取报文信息
12      {
13          gramBytes[i] = m_Data[m_dwDataGramHead+i];
14      }
15      m_Data.RemoveAt(0, m_dwDataGramTail);//将处理的报文数据从缓冲区中移除
16      this->PostMessage(WM_DECODE_MSG,(DWORD)(BYTE*)gramBytes,
17          (DWORD)(m dwDataGramLen));               //发送解码消息
18      m_dwDataGramHead=0;                           //赋值报文头变量
19      m_dwDataGramTail=0;                           //赋值报文尾变量
20      m_dwDataGramLen=0;                            //赋值报文长度变量
21      return;
22  }
```

4．数据解码OnDecodeMsg()函数

该函数实现协议数据包的解码功能。当程序提取出协议数据包后，发送消息给此解码函数，此函数即对数据进行解码，按照 NEMA 的协议对数据进行解析。该函数的传入参数为协议数据包的数据缓冲区指针和协议数据包的数据长度。该函数没有返回值。其代码如下：

```
01  void CDataFlow::OnDecodeMsg(DWORD dwData,
02      DWORD dwLen)//解码处理函数
```

```
03  {
04      //如果数据长度小于语句最小长度，则返回
05      if (dwLen < MAX_NEMA_SENTENCE_MIN_LENGTH)
06          return;
07      if (!dwData)
08          return;
09      GPSPack* pack=NULL;                     //解析后的数据包
10      //将协议数据放入数组中
11      BYTE gramBytes[MAX_NEMA_SENTENCE_MAX_LENGTH+1];
12      memset(gramBytes, 0x00, sizeof(gramBytes));
13      memcpy((void*)gramBytes, (void*)dwData, dwLen);
14      //判断数据尾是否正确
15      if ((gramBytes[dwLen-1] != NEMA_TAIL_STANDAD_2)
16          && (gramBytes[dwLen-2] != NEMA_TAIL_STANDAD_1))
17      {
18          return;
19      }
20      if (gramBytes[0] == NEMA_HEAD_TEXT) //处理 NEMA 文本格式的数据包
21      {
22          CString content;
23          content = gramBytes;
24          pack = DecodeNEMA_Text(content);
25      }
26      else if (gramBytes[0] == NEMA_HEAD_STANDAD)//处理 NEMA 语句
27      {
28          //判断逆数第五位是否为校验码，如果正确则将其取出
29          if (gramBytes[dwLen-5] != NEMA_CHECK_STANDAD)
30              return;
31          CString* packCheck=new CString();
32          packCheck->Format("%c%c", gramBytes[dwLen-4],
33              gramBytes[dwLen-3]);
34          //将数据分割成数组
35          CString m_item;
36          m_dataArray.RemoveAll();
37          for (DWORD i = 0;i < dwLen-NEMA_TAIL_LENGTH;i++)
38          {
39              if (gramBytes[i] == NEMA_SPLIT)
40              {
41                  CString* inArray = new CString();
42                  inArray->Format("%s", m_item);
43                  m_item.Empty();
44                  m_dataArray.Add(inArray);
45              }
46              else
47              {
48                  m_item += gramBytes[i];
49              }
50          }
51          CString* inLast = new CString();
52          inLast->Format("%s", m_item);
53          m_dataArray.Add(inLast);
54          if (m_dataArray.GetSize() < 1)
55              return;
56          //如果数据数组个数小于 1，则返回
57          if (!CheckData(packCheck))
58              return;   //校验数据，判断数据是否正确
59          m_dataArray.Add(packCheck);       //将校验码添加到数据数组中
60          CString type = *(CString*)m_dataArray.GetAt(0);//取出语句名
61          type = type.Mid(1);
```

```
62          //处理标准 NEMA 语句，以$GP 开头
63          if ((gramBytes[1] == NEMA_STANDAD_HEAD_1)
64              && (gramBytes[2] == NEMA_STANDAD_HEAD_2))
65          {
66              if (type == NEMA_STANDARD_GPGGA)          //非差分定位
67              {
68                  pack = DecodeNEMA_GPGGA();
69              }
70              else if (type == NEMA_STANDARD_GPGSA)     //当前卫星信息
71              {
72                  pack = DecodeNEMA_GPGSA();
73              }
74              else if (type == NEMA_STANDARD_GPGSV)     //可见卫星信息
75              {
76                  pack = DecodeNEMA_GPGSV();
77              }
78              else if (type == NEMA_STANDARD_GPRMC)     //推荐定位信息
79              {
80                  pack = DecodeNEMA_GPRMC();
81              }
82              else if (type == NEMA_STANDARD_GPVTG)     //地面速度信息
83              {
84                  pack = DecodeNEMA_GPVTG();
85              }
86              else if (type == NEMA_STANDARD_GPGLL)     //定位地理信息
87              {
88                  pack = DecodeNEMA_GPGLL();
89              }
90              else if (type == NEMA_STANDARD_GPZDA)     //时间和日期信息
91              {
92                  pack = DecodeNEMA_GPZDA();
93              }
94              else if (type == NEMA_STANDARD_GPDTM)     //大地坐标系信息
95              {
96                  pack = DecodeNEMA_GPDTM();
97              }
98          }
99          //高明公司的扩展协议
100         else if ((gramBytes[1] == NEMA_GARMIN_HEAD_1)
101             && (gramBytes[2] == NEMA_GARMIN_HEAD_2)
102             && (gramBytes[3] == NEMA_GARMIN_HEAD_3)
103             && (gramBytes[4] == NEMA_GARMIN_HEAD_4))
104         {
105             if (type == NEMA_USER_GARMIN_PGRME)       //估计误差信息
106             {
107                 pack = DecodeNEMA_PGRME();
108             }
109             else if (type == NEMA_USER_GARMIN_PGRMF)      //GPS 定位信息
110             {
111                 pack = DecodeNEMA_PGRMF();
112             }
113             else if (type == NEMA_USER_GARMIN_PGRMM)      //坐标系统信息
114             {
115                 pack = DecodeNEMA_PGRMM();
116             }
117             else if (type == NEMA_USER_GARMIN_PGRMT)      //工作状态信息
118             {
119                 pack = DecodeNEMA_PGRMT();
120             }
```

```
121             else if (type == NEMA USER GARMIN PGRMV) //三维速度信息
122             {
123                 pack = DecodeNEMA PGRMV();
124             }
125             else if (type == NEMA_USER_GARMIN_PGRMB) //信标差分信息
126             {
127                 pack = DecodeNEMA PGRMB();
128             }
129         }
130     }
131     //如果正确解析了数据，则发送消息给框架，做处理，如记录、在地图上显示等
132     if (pack != NULL)
133     {
134         ::SendMessage(AfxGetApp()|GetMainWnd()
135             |m_hWnd, WM_DECODED_NEMA_SENTENCE,
136             (WPARAM)&m_Com, (LPARAM)pack);
137     }
138 }
```

5．解码GPS定位信息数据包的DecodeNEMA_GPGGA()函数

该函数实现 GPS 定位信息协议数据包的解码功能，按照 NEMA0183 的$GPGGA 语句的协议格式对数据进行解析。该函数没有传入参数。该函数的返回值为 GPSPack 共用体指针，其中存储着 GPGGA 语句的数据项。其代码如下：

```
01  GPSPack* CDataFlow::DecodeNEMA_GPGGA()      //解析 GPGGA 协议
02  {
03      int total=16;                            //定义数据项个数
04      //判断当数据包中的数据项少于协议规定的 15 项时，则返回
05      if (m dataArray.GetSize() < total)
06          return NULL;
07      GPSPack* pack = new GPSPack();           //定义数据变量
08      memset(pack, 0x00, sizeof(GPSPack));
09      //为数据包的包头和包类型赋值
10      Pack->GPS NEMA GPGGA.head = NEMA HEAD STANDAD;
11      Pack->GPS NEMA GPGGA.bodyType = NEMATYPE STANDARD GPGGA;
12      //定义解析数据时用到的变量
13      int itemCount;                           //当前解析的是第几项
14      int itemLen;                             //当前解析项的长度
15      int iLen = 0;                   //当前解析项从第几个字符开始读取
16
17      //GPS 定位信息
18      //$GPGGA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,
19      //<9>,M,<10>,M,<11>,<12>*hh<CR><LF>
20      for (itemCount = 1; itemCount <total; itemCount++)
21          //从第一项依次解析每个数据项
22      {
23          iLen = 0;                   //设置每项从数据的第一个字符开始处理
24          CString* item = m_dataArray.GetAt(itemCount);
25          //并从数组中取出要解析的数据项
26          switch (itemCount)          //开始逐项解析数据项
27          {
28          case 1:
29              //时间部分     <1> UTC 时间，hhmmss（时分秒）格式
30              //时                2      UTC 时， "00".."23"
31              itemLen = 2;
32              CGPSPublic::ConvertToInt(
```

```
33                  (int&)pack->GPS NEMA GPGGA.body.time.hour,
34                  Item->Mid(iLen, itemLen));
35              //分                    2       UTC 分, "00".."59"
36              iLen += itemLen;
37              CGPSPublic::ConvertToInt(
38                  (int&)pack->GPS_NEMA_GPGGA.body.time.minute,
39                  item->Mid(iLen, itemLen));
40              //秒                    2       UTC 秒, "00".."59"
41              iLen += itemLen;
42              CGPSPublic::ConvertToInt(
43                  (int&)pack->GPS NEMA GPGGA.body.time.second,
44                  item->Mid(iLen, itemLen));
45              break;
46          case 2:
47              //纬度坐标<2> 纬度 ddmm.mmmm（度分）格式（前面的 0 也将被传输）
48              CGPSPublic::ConvertToLatitudeHavePoint(
49                  (double&)pack->
50                  |GPS_NEMA_GPGGA.body.latitude.latitude,*item);
51              break;
52          case 3:
53              //纬度半球<3> 纬度半球 N（北半球）或 S（南半球）
54              itemLen = 1;
55              CGPSPublic::ConvertToChar(
56                  (char&)pack->GPS NEMA GPGGA.body.latitudeType,
57                  *item->Mid(iLen, itemLen), itemLen);
58              break;
59          case 4:
60              //经度坐标<4> 经度 dddmm.mmmm（度分）格式（前面的 0 也将被传输）
61              CGPSPublic::ConvertToLongitudeHavePoint(
62                (double&)pack->GPS NEMA GPGGA.body.longitude.longitude,
63                  *item);
64              break;
65          case 5:
66              //经度半球      <5> 经度半球 E（东经）或 W（西经）
67              itemLen = 1;
68              CGPSPublic::ConvertToChar(
69                  (char&)pack->GPS_NEMA_GPGGA.body.longitudeType,
70                  *item->Mid(iLen, itemLen), itemLen);
71              break;
72          case 6:
73              //定位状态<6>GPS 状态：0=未定位，1=非差分定位，2=差分定位，6=正在
74              估算    itemLen = 1;
75              //CGPSPublic::ConvertToChar(
76                  (char&)pack->GPS NEMA GPGGA.body.gpsStatus,
77                  *item->Mid(iLen, itemLen), itemLen);
78              break;
79          case 7:
80              //卫星数量<7>
81              //正在使用解算位置的卫星数量（00～12）（前面的 0 也将被传输）
82              CGPSPublic::ConvertToInt(
83                   pack->GPS NEMA GPGGA.body.sateCount,
84                  *item);
85              break;
86          case 8:
87              //水平定位误差  <8> HDOP 水平精度因子（0.5～99.9）
88              CGPSPublic::ConvertToDouble(
89                   pack->GPS_NEMA_GPGGA.body.hdop,
```

```
90                  *item);
91              break;
92          case 9:
93              //海拔<9> 海拔高度（-9999.9～99999.9）
94              CGPSPublic::ConvertToDouble(
95                  pack->GPS_NEMA_GPGGA.body.altitude.altitude, *item);
96              break;
97          case 11:
98              //高度<10> 地球椭球面相对大地水准面的高度
99              CGPSPublic::ConvertToDouble(
100                 pack->GPS_NEMA_GPGGA.body.height,
101                 *item);
102             break;
103         case 13:
104             //差分时间<11>
105             //差分时间（从最近一次接收到差分信号开始的秒数，如果不是差分定位则
106             //为空）
107             CGPSPublic::ConvertToInt(
108                 pack->GPS_NEMA_GPGGA.body.diffSecond, *item);
109             break;
110         case 14:
111             //差分站编号<12>
112             //差分站 ID 号 0000～1023（前面的 0 也将被传输
113             //如果不是差分定位则为空）
114             if (item->|GetLength() > 0)
115             {
116                 itemLen = 4;
117                 CGPSPublic::ConvertToChar(
118                     (char&)pack->GPS NEMA GPGGA.body.diffStationID,
119                     *item->Mid(iLen, itemLen), itemLen);
120             }
121             break;
122         case 15: //hh 校验码
123             itemLen = 2;
124             CGPSPublic::ConvertToChar(
125                 (char&)pack->GPS_NEMA_GPGGA.check,
126                 item->Mid(iLen, itemLen), itemLen);
127             break;
128         }
129     }
130     //设置数据包的数据尾和协议校验码开始符
131     pack->GPS NEMA GPGGA.checkBegin = NEMA CHECK STANDAD;
132     pack->GPS_NEMA_GPGGA.tail[0] = NEMA_TAIL_STANDAD_1;
133     pack->|GPS NEMA GPGGA.tail[1] = NEMA TAIL STANDAD 2;
134     return pack;     //返回解码后的数据包
135 }
```

29.3.9　多线程串口工作方式

为了提高程序处理多串口的效率，实例中采用了多线程的工作方式，即每打开一个串口，则启动一个线程工作区 CComWorkFlow，并且在 GpsServerDoc 中进行控制。其工作流程图如图 29-13 所示。

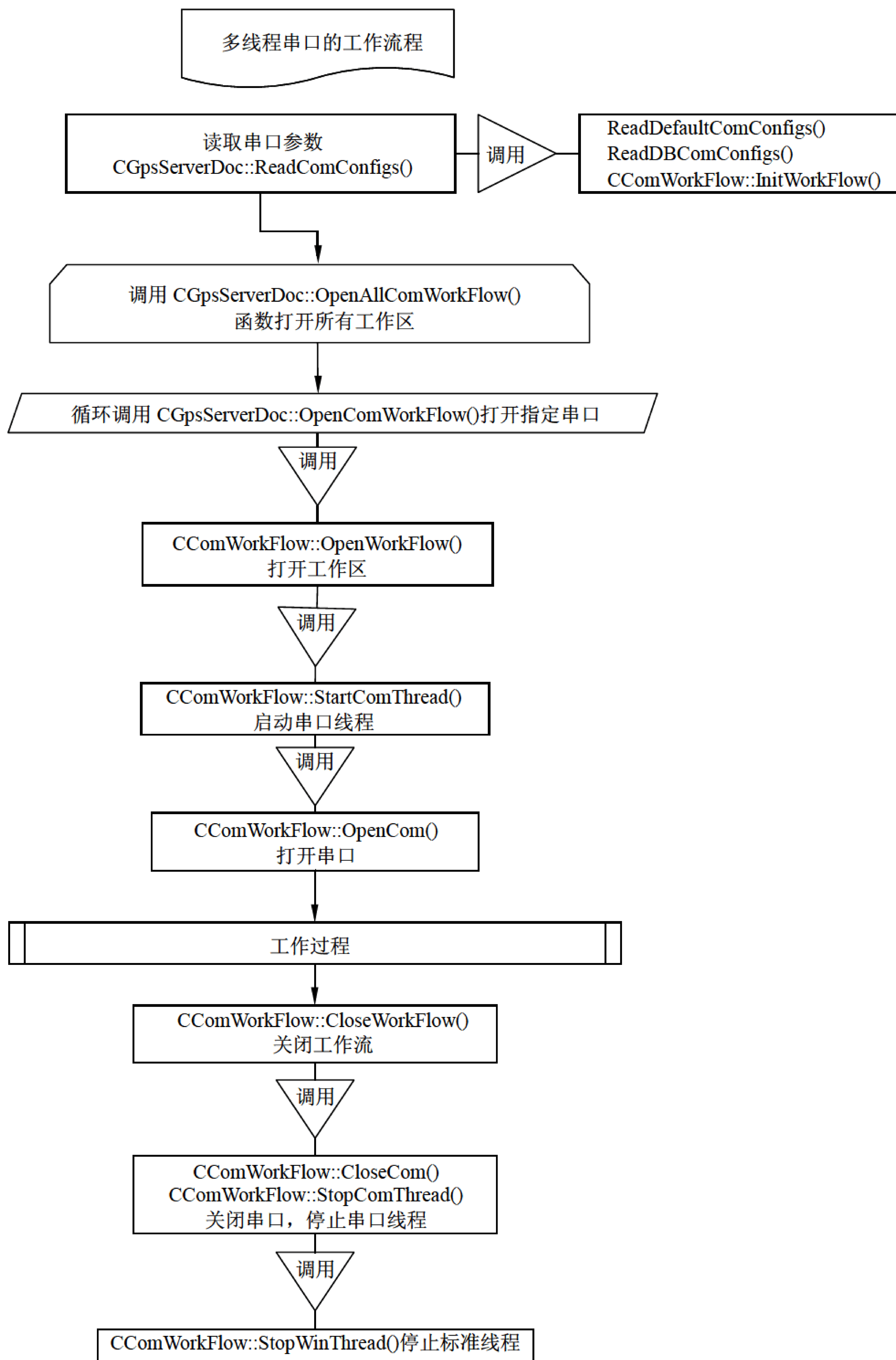


图 29-13　多串口工作流程

1．读取串口配置的ReadComConfigs()函数

该函数实现读取串口配置的功能。使用默认的串口配置（COM3，9600，8，0，0）。

该函数没有传入参数。该函数的返回值为 BOOL 型，表示读取串口配置是否正确。其代码如下：

```
01  BOOL CGpsServerDoc::ReadComConfigs()//读取串口配置
02       return ReadDefaultComConfigs();
03  }
```

2. 读取串口配置的ReadDefaultComConfigs()函数

该函数实现读取默认串口配置的功能。该函数没有传入参数。该函数的返回值为 BOOL 型，表示读取默认串口配置是否正确。其代码如下：

```
01  BOOL CGpsServerDoc::ReadDefaultComConfigs()//读取默认的串口配置信息
02  {
03      CMainFrame* pFrame = GetMainFrame();    //获取主对话框
04      m ComWorkFlow.RemoveAll();              //初始化串口参数链表
05      CComWorkFlow* pFlow = new CComWorkFlow();//设置默认的串口
06      if (pFlow==NULL)
07          return false;                       //如果初始化失败，则返回
08      DCB dcb;                                //定义串口参数变量
09      dcb.BaudRate = COM_BAUD_RATE;           //赋值波特率
10      dcb.ByteSize = COM DATA BYTE SIZE;      //赋值数据位数
11      dcb.Parity  = COM_PARITY;               //赋值奇偶校验
12      dcb.StopBits = COM_STOP_BIT;            //赋值停止位
13      pFlow->InitWorkFlow(COM_NAME, dcb);     //初始化工作流对象
14      m_ComWorkFlow.Add(pFlow);         //将工作流对象添加到串口工作流数组中
15      if (pFrame != NULL)                     //记录日志，输出提示信息
16          pFrame->WriteLog(LOG LEVEL PROMPT,
17          "读取默认串口信息----名称--%s;波特率--%d;
18          数据位--%d;奇偶校验--%d;停止位--%d;",
19          pFlow->m Com, pFlow->m dcb.BaudRate,
20          pFlow->m dcb.ByteSize,
21          pFlow->m dcb.Parity, pFlow->m dcb.StopBits);
22      return true;                            //函数返回
23  }
```

3. 获取串口工作流的GetComWorkFlow()函数

该函数实现根据串口的名称获取串口工作流变量的功能。该函数的传入参数为 CString 类型的串口名称。该函数的返回值为 CComWorkFlow 指针类型，表示对应的串口工作流指针。其代码如下：

```
01  CComWorkFlow* CGpsServerDoc::GetComWorkFlow(CString com)
02  {
03      //依次循环判断工作流
04      for (int i = 0; i < m ComWorkFlow.GetSize(); i++)
05      {
06          CComWorkFlow* pFlow = (CComWorkFlow*)m_ComWorkFlow.GetAt(i);
07          //获取第 i 个对象
08          if (pFlow != NULL)                //如果工作流对象不为 NULL
09          {
10              //如果工作流对象的串口名称与传入的相同，则返回
11              if (pFlow->m_Com == com)
12              {
13                  return pFlow;
```

```
14              }
15          }
16      }
17      return NULL;                            //查找不到，则返回 NULL
18  }
```

4．打开/关闭所有串口工作流的函数

该函数实现打开/关闭所有串口工作流的功能。读取串口配置后，程序调用此函数打开/关闭所有串口工作区。该函数没有传入参数。该函数的返回值为 BOOL 型，表示打开/关闭所有串口工作流是否成功。其代码如下：

```
01  BOOL CGpsServerDoc::OpenAllComWorkFlow()        //打开所有的工作流对象
02  {
03      //依次循环处理工作流
04      for (int i = 0;i < m_ComWorkFlow.GetSize(); i++)
05      {
06          //获取第 i 个对象
07          CComWorkFlow* pFlow = m_ComWorkFlow.GetAt(i);
08          if (pFlow == NULL)
09              continue; //如果工作流为 NULL，则跳转到下一个循环
10          OpenComWorkFlow(pFlow->m_Com);              //否则，打开工作流
11          if (i == 0)
12          {
13              CMainFrame* pFrame =  GetMainFrame();//获取主对话框
14              //将串口信息显示在界面上
15              if (pFrame != NULL)
16                  pFrame->AddViewComData(pFlow->m_Com);
17          }
18      }
19      return true;                                //函数返回 true
20  }
21  BOOL CGpsServerDoc::CloseAllComWorkFlow()       //关闭所有的工作流对象
22  {
23      //依次循环处理工作流
24      for (int i = 0;i < m_ComWorkFlow.GetSize(); i++)
25      {
26          CComWorkFlow* pFlow = m_ComWorkFlow.GetAt(i);//获取第 i 个对象
27          if (pFlow == NULL)
28              continue; //如果工作流为 NULL，则跳转到下一个循环
29          CloseComWorkFlow(pFlow->m_Com);             //关闭工作流对象
30      }
31      return true;                                //函数返回 true
32  }
```

5．打开/关闭指定串口的串口工作区的函数

该函数实现打开/关闭指定串口的串口工作区的功能。该函数的传入参数为 CString 类型的要打开/关闭的串口名称。该函数的返回值为 BOOL 型，表示打开/关闭指定串口的串口工作区是否成功。其代码如下：

```
01  //打开指定串口的工作流
02  BOOL CGpsServerDoc::OpenComWorkFlow(CString com)
03  {
04      CMainFrame* pFrame =  GetMainFrame();           //获取主对话框
```

```
05      //获取指定串口的工作流
06      CComWorkFlow* pWorkFlow = GetComWorkFlow(com);
07      if (pWorkFlow == NULL)          //如果不存在指定串口的工作流，则退出
08      {
09          return false;
10      }
11      BOOL bResult = pWorkFlow->OpenWorkFlow();//否则，打开工作流
12      if (bResult)                    //如果成功，则记录操作记录
13      {
14          if (pFrame != NULL)
15              pFrame->WriteComWorkFlowLog(pWorkFlow, true);
16      }
17      return bResult;                 //函数返回操作结果
18  }
19  //关闭指定串口的工作流
20  BOOL CGpsServerDoc::CloseComWorkFlow(CString com)
21  {
22      //获取指定串口的工作流
23      CComWorkFlow* pWorkFlow = GetComWorkFlow(com);
24      if (pWorkFlow == NULL)          //如果不存在指定串口的工作流，则退出
25      {
26          return false;
27      }
28      BOOL bResult = pWorkFlow->CloseWorkFlow();//否则，关闭工作流
29      if (bResult)                              //如果成功，则记录操作记录
30      {
31          CMainFrame* pFrame = GetMainFrame(); //获取主对话框
32          if (pFrame != NULL)
33              pFrame->WriteComWorkFlowLog(pWorkFlow,false);
34      }
35      return bResult;                           //函数返回操作结果
36  }
```

6. 启动/停止串口线程的函数

该函数实现启动/停止串口线程的功能，所做的工作是启动/停止串口接收数据的监测线程，打开/关闭串口。该函数没有传入参数。该函数的返回值为 BOOL 型，表示启动/停止串口线程是否成功。其代码如下：

```
01  BOOL CComWorkFlow::StartComThread()          //启动串口线程
02  {
03      CMainFrame* pFrame = (CMainFrame*)AfxGetApp()->GetMainWnd();
04      //获取主对话框
05      BOOL bResult = false;                     //定义返回值变量
06      TRY
07      {
08          if (m_ThreadCom != NULL)              //如果串口线程不为 NULL
09          {
10              if (m_ThreadCom->m_bOpen)         //判断串口线程是否打开
11              {                                 //输出提示信息
12                  if (pFrame != NULL)
13                      pFrame->WriteLog(LOG_LEVEL_INFO,
14                      "串口%s 已经打开!",m_Com);
15                  return false;
16              }
17          }
```

```
18          //启动串口工作线程
19          m ThreadCom =
20              (CThreadCom*)AfxBeginThread(
21              RUNTIME_CLASS(CThreadCom));
22          bResult = OpenCom();                         //打开串口
23          if (pFrame != NULL)                          //输出提示信息
24              pFrame->WriteLog(LOG LEVEL INFO,
25              "启动串口%s 工作线程成功!",m_Com);
26      }
27      CATCH(CException,e)                              //捕获异常
28      {
29          e->ReportError();                            //报告错误信息
30          m_ThreadCom->m_bDone=true;               //赋值线程终止变量为 true
31          //停止线程
32          StopWinThread((CWinThread*)&m_ThreadCom,INFINITE);
33          if (pFrame != NULL)                          //输出错误信息
34              pFrame->WriteLog(LOG_LEVEL_ERROR,
35              "启动串口%s 工作线程错误!", m_Com);
36          bResult = false;                             //赋值返回值为 false
37      }
38      END_CATCH
39          return bResult;                              //函数返回
40  }
41  BOOL CComWorkFlow::StopComThread()                   //停止串口线程
42  {
43      if (m_ThreadCom != NULL)                         //如果串口线程不为 NULL
44      {
45          m ThreadCom->CloseCom();                     //关闭串口
46          m_ThreadCom->m_bDone=true;               //赋值串口终止变量为 true
47          //停止串口线程
48          StopWinThread((CWinThread*)m_ThreadCom,INFINITE);
49          m_ThreadCom = NULL;                      //赋值串口线程对象为 NULL
50      }
51      return true;                                     //函数返回
52  }
```

7. 停止线程函数StopWinThread()

该函数实现停止线程的功能。该函数将停止标准的 CWinThread 线程的功能封装在一起。该函数的传入参数为要停止的线程的指针和停止的超时时间。该函数的返回值为 DWORD 型，表示停止线程的退出代码。其代码如下：

```
01  DWORD CComWorkFlow::StopWinThread(
02      CWinThread *pThread, DWORD dwTimeout)
03  {
04      //停止线程函数
05      if (pThread==NULL)
06          return NULL;                             //如果线程为 NULL，则返回
07      pThread->PostThreadMessage(WM_QUIT,0,0);//发送线程退出消息
08      ::WaitForSingleObject(pThread->m_hThread,dwTimeout);
09      //等待退出事件
10      DWORD nExitCode=0;
11      BOOL bFlag=::GetExitCodeThread(pThread->m hThread,&nExitCode);
12      //获取退出码
13      if (bFlag)                                   //如果退出，则删除线程句柄
14      {
```

```
15          delete pThread;
16      }
17      return nExitCode;                              //返回线程退出码
18  }
```

8．打开/关闭串口工作区的函数

该函数实现打开/关闭串口工作区的功能。其工作是启动/停止串口工作线程。该函数没有传入参数。该函数的返回值为 BOOL 型，表示打开/停止串口工作区是否成功。其代码如下：

```
01  BOOL CComWorkFlow::OpenWorkFlow()             //打开线程工作流
02  {
03      return StartComThread();                   //返回启动串口线程的返回值
04  }
05  BOOL CComWorkFlow::CloseWorkFlow()            //关闭线程工作流
06  {
07      return StopComThread();                    //返回停止串口线程的返回值
08  }
```

9．初始化串口工作区的InitWorkFlow()函数

该函数实现初始化串口工作区参数的功能。该函数的传入参数分别为串口名称和 DCB 结构的串口工作参数。该函数的返回值为 BOOL 型，表示初始化串口工作区是否成功。其代码如下：

```
01  //初始化线程工作流
02  BOOL CComWorkFlow::InitWorkFlow(CString com, DCB dcb)
03  {
04       m_Com = com;                                      //赋值串口名称
05       m_dcb = dcb;                                      //赋值串口参数
06       m_DataFlow->Create();                             //创建串口工作流
07       m_DataFlow->SendMessage(WM_INITCENTER,NULL,(LPARAM)0);
08                                                         //发送数据流消息
09       return true;                                      //返回
10  }
```

10．打开/关闭串口的函数

该函数实现打开/关闭串口的功能。该函数没有传入参数。该函数的返回值为 BOOL 型，表示打开/关闭串口是否成功。其代码如下：

```
01  BOOL CComWorkFlow::OpenCom()             //打开串口函数
02  {
03      if (m_ThreadCom == NULL)
04          return false;     //如果串口线程为 NULL，则返回
05      CMainFrame* pFrame = (CMainFrame*)AfxGetApp()->GetMainWnd();
06      //获取主对话框
07      if (m_ThreadCom->m_bOpen)        //如果串口已经打开，则输出提示信息
08      {
09          if (pFrame != NULL)
10              pFrame->WriteLog(LOG_LEVEL_INFO,
11              "串口%s 已经打开!", m_Com);
12          return false;
13      }
14      if (m_ThreadCom->OpenCom(m_Com, m_dcb,m_DataFlow,
```

```
15          m DataFlow->GetMeMsg()))
16      {                                          //打开串口，并输出提示信息
17          if (pFrame != NULL)
18              pFrame->WriteLog(LOG_LEVEL_INFO,
19              "打开串口%s 成功!", m_Com);
20          m_DataFlow->m_Com = m_Com;
21          return true;
22      }
23      Else                                       //如果失败，则输出错误提示
24      {
25          if (pFrame != NULL)
26              pFrame->WriteLog(LOG_LEVEL_ERROR,
27              m_ThreadCom->m_sError+
28              ",请重新配置串口!");
29          m ThreadCom->m bOpen=false;
30          return false;
31      }
32  }
33  BOOL CComWorkFlow::CloseCom()              //关闭串口处理函数
34  {
35      if (m_ThreadCom == NULL)
36          return false;
37      //如果串口线程为 NULL，则返回 false
38      return m ThreadCom->CloseCom();        //否则关闭串口
39  }
```

29.3.10　发送命令

虽然在 GPS 信息接收程序中可以不使用向串口发送数据的功能，但是前面介绍过，当 GPS 模块没有准备好时，可以通过发送模拟数据来测试。Win32 下向串口发送数据使用 WriteFile()函数来实现。在发送数据时，需要注意发送操作的异步处理。在本实例中发送命令通过 SendData()函数来实现。此函数实现向串口发送数据的功能。其传入参数为要发送的数据的缓冲区指针和要发送的数据长度。其返回值为 BOOL 型，表示发送数据是否成功，成功则返回 true，失败则返回 false。具体代码如下：

```
01  BOOL CThreadCom::SendData(LPCTSTR sBuf, DWORD dwLen)   //发送数据
02  {
03      BOOL bResult = false;                      //定义函数返回值变量
04      if (m_hCom != INVALID_HANDLE_VALUE)        //判断串口句柄是否有效
05      {
06          DWORD    dwError;                      //发送操作时的错误代码
07          DWORD        dwErrorFlags;             //发送操作时的错误状态
08          DWORD        dwByteSent=0;             //发送的字符数
09          DWORD        dwByteWrite;              //发送字符数临时变量
10          COMSTAT  ComStat;                      //串口的工作状态
11          //向串口发送数据，需要注意 m_overWrite 的使用，这是执行异步写操作的关键
12          BOOL fWriteStat = WriteFile(m hCom, sBuf, dwLen,
13              &dwByteWrite, &m overWrite);
14          if (!fWriteStat)      //如果发送数据发生错误，则处理判断错误的情况
15          {
16              dwError = GetLastError();          //获取错误代码
17              //如果错误是异步读写未完成，则继续等待发送
18              if(dwError == ERROR IO PENDING)
19              {                                  //首先应该判断发送是否完成
```

```
20                  while(!GetOverlappedResult( m_hCom,&m_overWrite,
21                      &dwByteWrite, true ))
22                  {
23                      dwError = GetLastError();
24                      if(dwError == ERROR_IO_INCOMPLETE)
25                      {                //如果发送还没有完成，则继续等待发送结束
26                          dwByteSent += dwByteWrite;
27                          continue;
28                      }
29                      else                           //如果发生错误，则处理错误
30                      {
31                          m_sError.Format("<%u>", dwError) ;
32                          ClearCommError(m_hCom,
33                              &dwErrorFlags, &ComStat);
34                          if (dwErrorFlags > 0)
35                          {
36                              printf(m_sError, "%s<%u>",
37                                  m_sError,dwErrorFlags);
38                          }
39                          break;
40                      }
41                  }
42                  //累计增加发送成功的字符数，并根据情况编写信息提示
43                  dwByteSent += dwByteWrite;
44                  if( dwByteSent != dwLen ) //输出发送的信息
45                  {
46                      printf(m_sError,
47                          "%s-发送超时：%ld/%ld 已经发送/共有字节",
48                          m_sError, dwByteSent, dwLen);
49                      bResult = false;
50                  }
51                  else
52                  {
53                      printf(m_sError,
54                          "%s-发送完成：共成功发送%ld 个字节", m_sError,
55                          dwByteSent);
56                      bResult = true;
57                  }
58              }
59              else                                  //如果发生其他错误，则处理错误
60              {
61                  m_sError.Format("<%u>", dwError) ;
62                  ClearCommError( m_hCom, &dwErrorFlags, &ComStat ) ;
63                  if (dwErrorFlags > 0)
64                  {
65                      printf(m_sError, "%s<%u>", m_sError, dwErrorFlags);
66                  }
67                  bResult = false;
68              }
69          }
70          else                                      //如果发送成功，直接编写信息提示
71          {
72              printf(m_sError,"发送完成：共成功发送%ld 个字节",dwByteSent);
73              bResult = true;
74          }
75      }
76      else                                          //如果串口句柄无效，则返回错误
77      {
78          m_sError="串口句柄无效";
```

```
79            return false;
80        }
81        //将发送结果发送给主对话框，由主对话框处理界面显示的问题
82        ::SendMessage(AfxGetApp()->GetMainWnd()->m_hWnd,
83            WM_SEND_COM_DATA_RESULT,
84            (WPARAM)&m_Com, (LPARAM)&m_sError);
85        return bResult;                       //函数返回
86    }
```

29.3.11　结束清理

当串口工作结束时，需要关闭串口并释放相关资源。在实际编写程序时，往往会忽略了结束清理工作。实际上，没有结束清理处理的程序不是一个完整的程序。尤其是像串口这样独享的资源，一定要做清理释放工作。在本实例中，清理释放工作就是关闭串口，通过 CloseCom()函数来实现。该函数清除串口工作区并关闭串口句柄。该函数没有传入参数。该函数的返回值为 BOOL 型，表示关闭串口是否成功，返回 true 表示关闭串口成功，返回 false 表示关闭串口失败。具体代码如下：

```
01    BOOL CThreadCom::CloseCom()               //关闭串口
02    {
03        //如果串口句柄是有效的，则清除串口工作区，关闭句柄
04        if (m_hCom!=INVALID_HANDLE_VALUE)    //判断串口句柄是否有效
05        {
06            PurgeComm(m_hCom,PURGE_RXCLEAR);//清除缓冲区中的数据
07            CloseHandle(m_hCom);             //关闭串口句柄
08            m_hCom=INVALID_HANDLE_VALUE;      //赋值串口句柄
09        }
10        m_bOpen=false;                        //赋值串口是否打开变量为 false
11        return true;                          //函数成功返回
12    }
```

29.3.12　地图支持

在解析出 GPS 信息后，如果是枯燥的数据，对用户不够直观，无法确切地了解实际的位置。这就需要加入对地图的支持。

地图的应用是非常复杂的，属于 GIS（地理信息系统）技术。简单地讲，目前在程序中集成地图应用有两种方式，一种是本地地图应用，另一种是 Web 地图应用。最早地图的应用是从本地应用发展起来的，其技术已经比较成熟了，像 MapInfo 的 MapX、Arc/Info 等都提供了本地地图的使用。随着本地地图应用的一些问题，如地图费用、占用空间大、部署麻烦等日趋明显化，出现了 Web 地图的应用。在本实例中，使用的就是 Web 地图应用技术。

目前比较流行的 Web 地图应用引擎提供商有 51ditu、MapABC、MapBar 和 GoogleEarth 等。本实例中，使用 51ditu 提供的免费地图引擎，实现 GPS 位置信息的图形化显示。地图显示采用的方法是，将 51ditu 引擎的 Web 页面编写好，主要是显示点的脚本函数，然后在程序中调用页面的脚本函数。Map51POI.html 文件的代码如下：

```
01    <html xmlns:v="urn:schemas-microsoft-com:vml">
02        <head>
```

```
03      <meta http-equiv="Content-Type"
04       content="text/html; charset=gb2312"/>
05      <meta name="keywords"
06       content="LTMarker,LTMaps.addOverLay,JavaScript,GPS,地图,
07                范例文档,vml"/>
08      <title>GPS 位置</title>
09      <style type="text/css">v\:*{behavior:url(#default#VML);}</style>
10      <script language="javascript"
11          src=http://api.51ditu.com/js/maps.js></script>
12  <script language="javascript">
13      var map;
14  function onload()                               //初始化地图
15  {
16      map=new LTMaps("mapDiv");                   //创建地图对象
17      map.centerAndZoom("青岛",6);                //初始化地图中心点和缩放等级
18      map.addControl(new LTStandMapControl());//增加地图控件
19  }
20  function showpoint(x, y, name)                  //显示位置点
21  {
22      map.clearOverLays();                        //清除当前地图上所有的图层对象
23      var point = new LTPoint( x , y);            //创建点对象
24      var marker1 = new LTMarker( point );        //创建标记对象
25      map.addOverLay( marker1 );                  //增加图层对象
26      var infoWin = new LTInfoWindow( marker1 );//创建信息窗口对象
27      infoWin.setLabel(name);                     //设置信息窗口的标题
28      map.addOverLay( infoWin );                  //将信息窗口增加到地图图层中
29      map.setCenterAtLatLng(point);               //将地图中心点移动到此位置
30  }
31  </script>
32      </head>
33      <body onLoad="onload()">
34      <div id="mapDiv"
35        style="position:absolute;width:100%; height:100%;">
36      <div align="center"
37        style="margin:12px;"><a href="http://api.51ditu.com/docs/
38      mapsapi/help.html"
39       target="_blank" style="color:#D01E14;font-weight:
40         bolder;font-size:12px;">看不到地图请点这里</a></div></div>
41      <script language="javascript"
42         src="http://api.51ditu.com/js/pv.js">
43      </script></body>
44      </html>
```

在程序中，要在地图上显示位置点，则需要以下函数的支持。

1．显示位置点的ShowPoint()函数

该函数实现显示位置点的功能。该函数的传入参数分别为位置点的经度、纬度和名称。该函数没有返回值。其代码如下：

```
01  void CGpsServerView::ShowPoint(double x, double y, CString name)
02                                                  //调用显示点的函数
03  {
04      CString js;                                 //定义命令变量
05      js.Format( T("showpoint(%f, %f, \"%s\");"), x, y, name);
06                                                  //格式化命令变量
```

```
07      ExecJavaScript(js);                         //执行 js 脚本
08  }
```

2．执行JavaScript的ExecJavaScript()函数

该函数实现在程序中调用 Web 页面中的 JavaScript 脚本的功能。该函数的传入参数为要执行的 JavaScript 脚本。该函数没有返回值。其代码如下：

```
01  void CGpsServerView::ExecJavaScript(CString js)//执行 js 脚本的函数
02  {
03      //获取 HTML 文档
04      CComQIPtr<IHTMLDocument2> pDoc = (IHTMLDocument2*)
05          GetHtmlDocument();
06      if (pDoc == NULL) return;
07      //获取 HTML 窗体
08      CComQIPtr<IHTMLWindow2> pWin;
09      pDoc->get parentWindow(&pWin);
10      if (pWin == NULL)
11          return;                     //判断返回结果
12      //执行 JavaScript 脚本
13      CComBSTR bstrJS = js.AllocSysString();      //定义执行的脚本命令
14      CComBSTR bstrLanguage = SysAllocString(L"javascript");
15      //定义执行的脚本语言
16      VARIANT varResult;                          //定义返回值变量
17      //执行 js 脚本
18      pWin->execScript(bstrJS, bstrLanguage, &varResult);
19  }
```

29.3.13　程序测试截图

经过了上面程序的支持，就可以实现 GPS 信息的接收了，图 29-14 和图 29-15 为程序的测试截图。

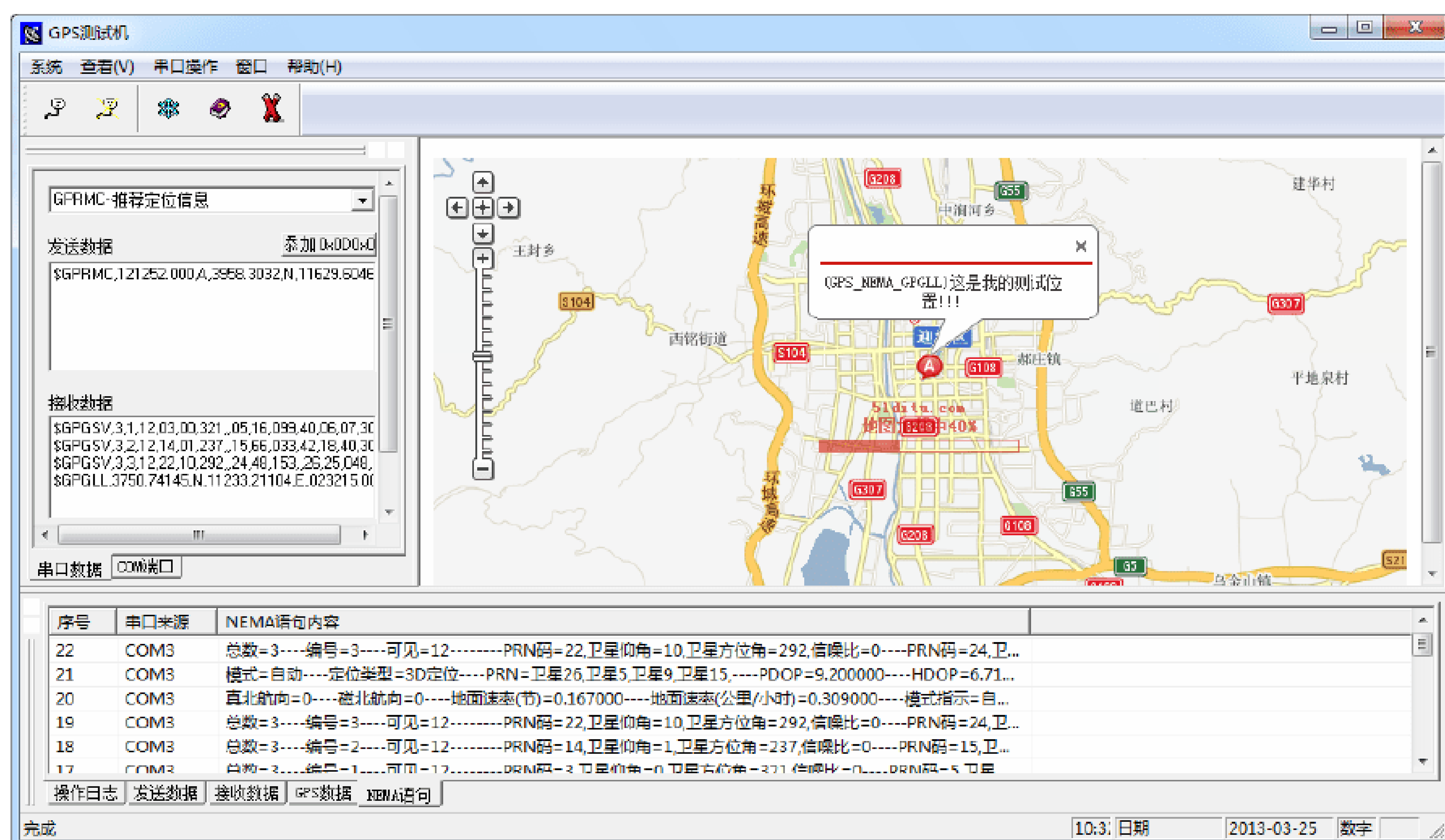


图 29-14　GPS 接收程序测试截图 1

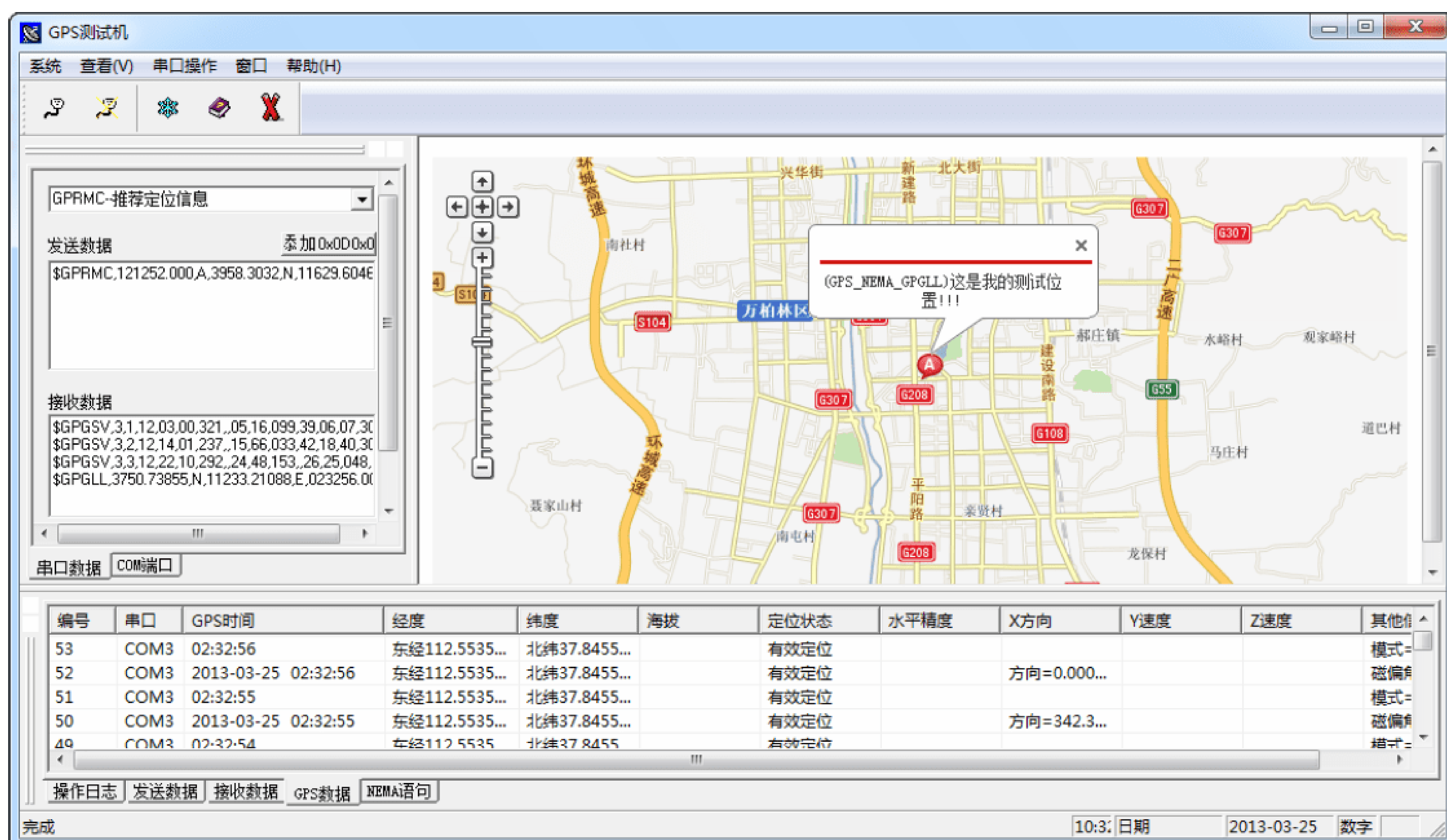


图 29-15　GPS 接收程序测试截图 2

29.4　本 章 小 结

本章通过一个实例，讲解了使用 Visual Studio 2010 开发通信程序的方法和步骤。本章重点是掌握串口编程的方法。本章的难点是如何编写高效的通信前置机。通过本章的学习，读者应该更加深入地理解串口的工作原理和串口程序的编写，以及多线程程序的控制。